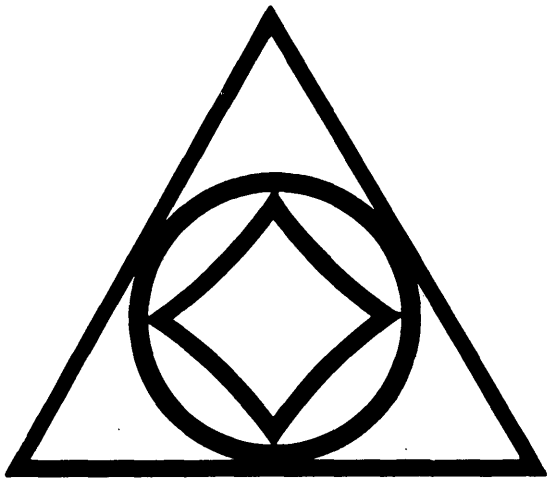


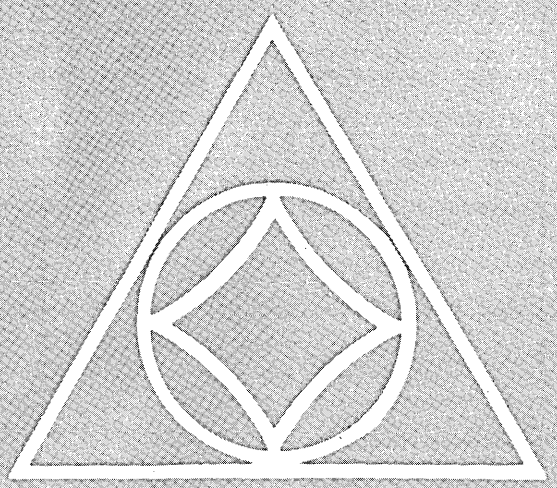
AFIPS

CONFERENCE PROCEEDINGS

VOLUME 31

1967

FALL JOINT COMPUTER CONFERENCE



AFIPS

CONFERENCE PROCEEDINGS

VOLUME 31

1967

FALL JOINT COMPUTER CONFERENCE

November 14 - 16, 1967
Anaheim, California

AFIPS PRESS
210 SUMMIT AVENUE
MONTVALE, NEW JERSEY 07645

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1967 Fall Joint Computer Conference Committee or the American Federation of Information Processing Societies.

Library of Congress Catalog Card Number 55-44701
AFIPS Press
210 Summit Avenue
Montvale, New Jersey 07645

CONTENTS

SPECIAL ACKNOWLEDGMENT

Data Processing Program for Technical Papers

To maintain good control over the status of the three hundred technical papers submitted for this conference from receipt through distribution to reviewers, return from reviewers, final review and selection, and final disposition, a special data processing program was written to keep track of status and issue timely status reports. Special acknowledgment is made of the work of Mrs. Bernice Bjerke, who wrote the program, and to Aerospace Corporation for underwriting the costs of writing, operating and documenting the system.

Multiprogramming for hybrid computation

by MARK S. FINEBERG and OMRI SERLIN
McDonnell Automation Company
Division of McDonnell Douglas Corporation
St. Louis, Missouri

INTRODUCTION

A significant recent development in hybrid computation is the increasing use of multiprogramming techniques and multiprocessing hardware.[1,2,3] To some extent this trend is motivated by the development of multi-user systems in the pure digital field. However, the primary justification for hybrid multiprogramming is economic. It is possible to show that, by sharing a large, powerful central facility, the cost-per-computation can be reduced by almost an order of magnitude, as compared with the alternative of using several smaller, wholly-committed computers.*

Associated with any multiprogramming system, hybrid or batch, are two fundamental requirements. There must be a mechanism through which the available resources of the system (processor(s), working storage, I/O) are allocated intelligently to satisfy users' needs while maintaining maximum throughput. Equally important is the need to guarantee the integrity of the system: users' programs must be protected from, and independent of, each other, and the operating system must be immune to interference from any and all users.

The presence of time-critical tasks in the job mix complicates both the resource allocation problem and the task of safeguarding the system's integrity. The system must not only provide *sufficient* services, but these services must be rendered *in time*, that is, within a firm time limit. It is also necessary to detect whenever a job is attempting to use more than its assigned share of resources, before this infringement affects other users. The software and hardware features that permit successful resolution of these requirements in a time-critical environment, are the subject of this paper.

A definition of hybrid multiprogramming

The concepts and terminology of hybrid multiprogramming are probably new, so it is worthwhile to establish a clear frame of reference at the outset.

What is a time-critical hybrid job?

As far as the digital computer is concerned, a time-critical hybrid job simply represents certain loads on the central processor (or other processors, if any), memory and I/O facilities. It does not matter much whether the hybrid nature of the program is due to the presence "in the loop" of analog computers, analog hardware, or both. What does matter is that the time-critical job must receive its share of the processor within a firm time limit. This time limit is shorter, by one or two orders of magnitude, than the response time in other "real-time" systems such as airline reservations or on-line banking. Another peculiarity of the time-critical hybrid job is that it will rarely, if ever, tolerate the sort of partial degradation ("overload") that is quite acceptable, though undesirable, in a commercial "real-time" environment. If, for any reason, a time-critical hybrid job fails to get its quota of processor time even once during a run—in one run the job may need the processor hundreds of thousands of times—that run must, in most cases, be abandoned or restarted.

There is another important aspect in which the hybrid job differs from others; "page leafing" techniques cannot be applied to it, nor can it be relocated occasionally in core, because the time frames within which it needs processor service are much too short to accommodate either of these operations in the present state of the art.

The goals of hybrid multiprogramming

To be classified as a hybrid multiprogramming system, a computing complex must be capable of simultaneously servicing two or more time-critical hybrid tasks, in addition to pure digital batch processing. The reason for insisting on a minimum of two hybrid jobs is not purely semantic: the requirements imposed on the system configuration to enable it to run two time-

*The debate between the proponents of these opposing viewpoints certainly deserves a wider recognition. For more detailed discussions, see references 5, 6, and in particular, 7.

critical tasks simultaneously are much more demanding than those for servicing only one such job, whereas the extension from two to any arbitrary number is relatively straight forward.

The batch load may represent overflow from a pure digital facility, or additional hybrid jobs in the digital checkout phase, or it may be the primary load of the system, the hybrid jobs being occasional users only.

In any case, the most important criterion by which the success of a hybrid multiprogramming system must be measured is the degree to which each hybrid user is made to feel that he has a committed computer at his disposal. To maintain this illusion it is necessary to guarantee *protection* and *independence* to all jobs. Protection requires that *errors committed by one program must not be allowed to interfere with the successful running of programs other than the offending one.* Independence implies that the *coding, loading* and *checkout* of one program need never be geared to the presence, absence or idiosyncrasies of other programs.

Almost equally important is the need to maintain maximum throughput while servicing job mixes with varying proportions of time-critical hybrid simulations, other real-time tasks, and batch jobs. The economic justification for hybrid multiprogramming depends to a large extent on the ability of the system to service I/O-bound batch jobs concurrently with the hybrid jobs, which make relatively light use of peripheral equipment but represent a heavy load on the processor. Consequently the system cannot be regarded as a highly successful one if the software features that are needed to serve time-critical tasks significantly degrade the throughput performance for all other jobs.

These are the ground rules; they need to be kept in mind in the following discussion.

The central processor allocation problem

The proper allocation of the central processing unit (CPU) in the presence of two or more time-critical jobs is of prime importance in a hybrid multiprogramming system. These time-critical jobs, from the standpoint of the CPU, are characterized by three parameters (Figure 1):

1 Repetition Period, or Frame Time (T_f). This is the period between two successive instants at which the job demands CPU action.
2 Response Time, or Tolerance (T_r). This is the period within which the equipment external to the digital computer must have a "reply." Tolerance is measured from the instant of CPU request.
3 Compute Time (T_c). This is the amount of CPU service, in terms of CPU-seconds, that the job needs when it asks for the CPU.

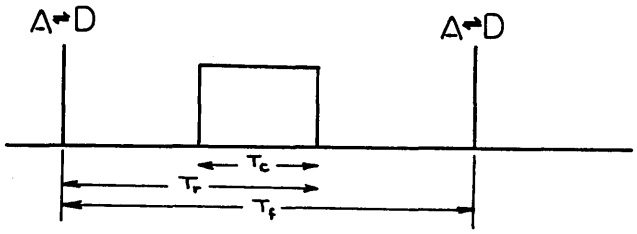


Figure 1—*The parameters of a hybrid job:* T_f is the repetition period, usually marked by clock pulses which cause A↔D transfer. T_r is the response time; in this figure the digital computer finishes its computation just in time to meet the response-time constraint. T_c is the execution time of the digital program.

Perhaps the most common hybrid job is the one for which all three parameters are constant (or nearly so) and $T_r = T_f$. Typically, an external "real time clock" is used to generate pulses at intervals T_f, (which are hence also called "clock periods"). At the instants marked by the real time clock pulses ("clock interrupts"), analog-to-digital (AD) conversion is initiated. At the same instant, digital-to-analog (DA) data, which is computed and stored in the DAC buffers during the previous clock period, is also converted. To account for the fact that the DA data represents results of computations based on conditions existing at the previous clock interrupt, the digital program normally includes a predictive algorithm that extrapolates the DA data to the time at which the data is actually transferred to the analog domain. Because of this extrapolation, the "external world" (analog domain) is insensitive to the actual time of execution of the digital step, as long as all the required computation is completed prior to the next clock pulse.

A somewhat more complicated type of job is characterized by $T_r < T_f$, all three parameters still being constant. This situation arises when equipment external to the digital computer requires some information, which is dependent on the CPU for its computation, sooner than the next clock pulse, when the normal AD transfer takes place. Still another possible configuration results when T_c is not a constant. This situation occurs when different paths are taken within the program in different clock periods.

The so-called "asynchronous interrupt" routine represents another, important form of a hybrid task. Typically, this job has constants T_r and T_c, but its repetition period T_f is variable.

The importance of these parameters is that they bring order to the analysis of situations that, on the surface, appear to be vastly different from each other. For example, the periodic hybrid job, with con-

stant T_r and T_c and $T_r = T_f$, can be regarded as a special case of the more general "asynchronous interrupt" job, with distinct (and possibly variable) T_f, T_r and T_c.

The two-job case

It should be obvious that a single time-critical job, all others being batch-type, can be accommodated quite simply by assigning to it the highest allowable priority. Of course we assume here that the worst-case monitor delay—the longest period during which the monitor is "blind" to CPU requests, plus the time it needs to decide if and what action to take, plus the actual program-switching time—is negligible in comparison to the job's tolerance. The simplest situation that is of interest is, therefore, that of the two time-critical jobs, all others being batch-type. It is convenient to assume initially that these jobs are of the simplest possible type, that is, they have constant (though not necessarily equal) repetition periods, and nearly constant compute time requirements and tolerances, the latter being equal to the respective repetition periods. As indicated earlier, such jobs are quite common.

If the two jobs use independent real time clocks, then conflicts in CPU demand are certain to occur. One way to avoid conflicts entirely is to prohibit the use of independent clocks, and this method (the "master schedule" approach) is discussed later. Otherwise, the monitor must resolve such conflicts. The process of resolving these conflicts is the essence of the CPU scheduling algorithm; this process can be termed "dynamic scheduling" to distinguish it from "static" scheduling tasks; that is, scheduling decisions that can be made at leisure relative to real-time requirements.

Fixed priority scheduling

Perhaps the simplest CPU scheduling is that in which the two time critical jobs are allocated absolute priorities at load time ("statically"). The job having the shorter clock period receives the higher priority. Thereafter, when conflicts occur, the CPU requests of the higher priority job are always honored before those of the one with the longer clock period. As jobs terminate and other time-critical jobs take their place, the clock periods are compared by the operating system and the priorities are readjusted accordingly.

It may not be obvious that this procedure can lead to difficulties; an illustration of such a case is given in Figure 2. As Figure 2 indicates, periods of relative congestion can occur, followed by periods in which the CPU is not needed by either of the time-critical jobs. These situations can be described compactly in terms of percent loading (PCL) of the individual jobs. PCL's are defined as the ratio of the required CPU time per clock period to that period; that is, $PCL = 100\ T_c/T_f$. In Figure 2, for instance, job A, which requires .5T CPU time every T seconds, has a PCL of 50%, while job B (.5T every 1.5T) has a PCL of 33.3%. The sum of the PCL's is well under 100%, so that in the absence of monitor delays and system overhead, it is tempting to conclude that the CPU requirements of both jobs can always be accommodated. Actually, as the figure illustrates, this is not the case; had job B needed 40% PCL, it could not have been satisfied in the first period (period I). In fact, it is possible to prove that, using the priority scheme described above, and assuming zero delays and no overhead, the CPU can be scheduled unconditionally among two time-critical jobs phose PCL's add up to 100% only if their clock periods are integer multiples of each other. Further, it is possible to show that, when the clock periods are not integer multiples, then the most difficult case to accommodate is the one in which the longer clock period is exactly $\sqrt{2}$ times the shorter period (see Figure 3); and in this case the best that can be done is to guarantee CPU service unconditionally only if the sum of the PCL's is less than $2(\sqrt{2}-1)\ 100$ or about 83%. This is by no means an impossible restriction; in fact, if the system is required to serve some batch jobs concurrently with the two time-critical ones, the latter must be restricted so that they do no monopolize the CPU to the exclusion of all other jobs. However, it is important to understand the source of this limitation and how it can be removed.

As can be seen in Figure 2, while the CPU is loaded to capacity in period I, there is more than enough CPU time available in the second period. What has happened is, therefore, a "local lockout" condition, in which the higher priority job temporarily "locked out" the lower priority one. The reason for this lockout is that job A did not really need the CPU right after its second clock interrupt: it could have waited as much as .5T after that clock and still have received .5T CPU time before its next clock (clock pulse 3). Thus the difficulty is simply due to an unintelligent allocation of the CPU, and that in turn can be traced to the manner in which the priorities were determined: fixed at load-time and based on the shortest clock period. In general, in the presence of independent, unsynchronized real time clocks, a static priority scheme determined at load-time or fixed by hardware cannot result in a completely efficient CPU allocation. A dynamic priority allocation scheme, based on the *relative urgencies* of CPU tasks, is more appropriate.

The relative urgency algorithm

Figure 4 shows the two job case (as in Figure 2) but with the CPU being allocated to the job that has the *least-time-to-go;* that is, the one whose next clock pulse is most imminent and that has not completed its com-

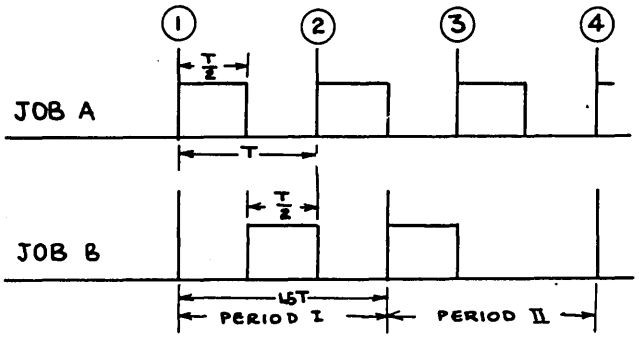


Figure 2—*CPU scheduling, two-job case:* Job A has the shorter period and, therefore, the highest absolute priority. During period I, job B can have no more than .5T CPU time (PCL—30%) while job A claims .5T (PCL—50%) twice; during the second (II) period, the CPU is idle (or serving batch jobs) for .5T.

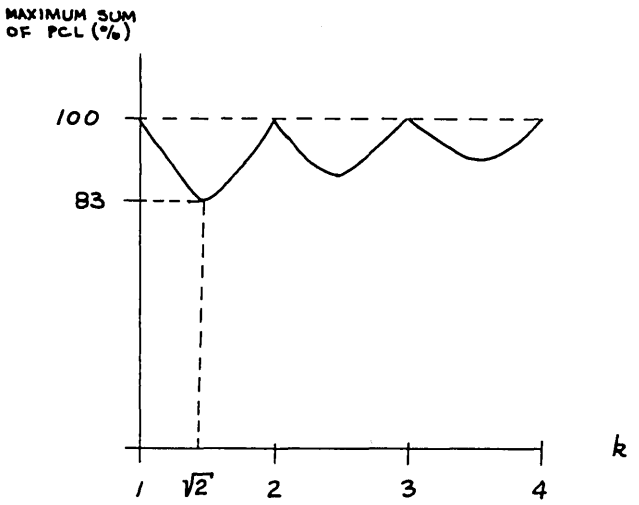


Figure 3—*The maximum sum of PCL's for various ratios of clock intervals:* The longer job's clock interval is k times the shorter interval. When k is an integer, the sum of the PCL's can add to 100%. When $n-1<k<n$, the worst case occurs at $K=\sqrt{n(n-1)}$; the maximum sum of PCL that can be scheduled is $\alpha + \beta$ where $\alpha = n + k$ and $\beta = n(1 - \alpha)/k$. (α and β are respectively PCL's of the shorter- and longer- clock-period jobs.)

puting tasks for its current clock period. This process is evident, for example, at clock pulse 2, where job A does not get the CPU since job B has the least-time-to-go. When the times-to-go for both jobs are equal their real priorities are also equal; it does not matter then who gets the CPU first. In such cases the decision can be based on nominal, or load-time, priorities if none of the requesting jobs is in possesion of the CPU; if the conflict occurs when one of the jobs with equal times-to-go is executing, that job retains control of the CPU, in order to minimize program-switching. Figure

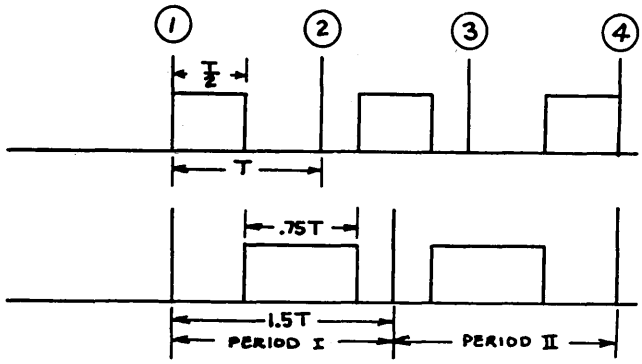


Figure 4—*CPU scheduling based on relative urgencies:* In case of conflict, the job whose clock pulse is most imminent receives the CPU, unless the job has finished its tasks for its current clock period.

4 shows how the "relative-urgency" algorithm is able to schedule, without lockouts, the CPU among two jobs whose PCL's add up to 100%.

The implementation of this algorithm is, in principle, surprisingly simple.* The monitor keeps a "time-to-go" table, or stack, which contains one slot for every time-critical job; into the slot the monitor inserts the declared clock period of the job whenever that job's clock interrupt occurs. The monitor also keeps running time (which it has to do in any case for accounting purposes and for the protection of the system). Whenever a clock interrupt occurs for any job, the monitor subtracts the elapsed time (measured from the last clock interrupt) from all entries in the stack. Then the monitor searches the stack for the smallest number and assigns the CPU to the corresponding job. When a job completes its assigned tasks for a given clock period, it signals the monitor and the latter either removes the job from the time-to-go stack, or inserts a very large number in that job's slot. Batch jobs can also be entered into the stack with very large (but finite) "time-to-go," the smaller numbers going to the higher priority batch jobs; or these numbers can actually represent "time-slices" to facilitate round-robin time-sharing of the batch jobs. As long as the batch jobs are never allowed to have shorter times-to-go than the time-critical ones, the former time-share the CPU whenever the latter do not require it.

The discussion in the preceding paragraph implies servicing of any number of time-critical jobs (rather than two) and batch jobs. This is not unintentional, since the mechanism described is indeed capable of

*The implementation described is used merely to explain the concept; many variations are possible that are better in one sense or another.

handling any number of time-critical jobs. For two such jobs, the time-to-go table contains possibly only two entries; more jobs require a larger table. If the table is physically arranged according to its contents—that is, highest priority (least time-to-go) on top, lowest priority on bottom, and all entries include job identification—then the sorting operation required when a clock interrupt occurs reduces to one comparison only: the clock period of the interrupting job, which at the interrupt instant equals its time-to-go, is compared with the top entry in the stack. If the interrupting job has less time-to-go than the one currently executing (top of the stack), the latter is removed from the top position; otherwise, there is no change in the top entry of the stack. The process of fitting either the interrupting job (or the one it displaced from the top of the stack) into the proper place in the stack can be carried out after the CPU has been switched (if needed), that is, relatively at leisure.** The stack is, in effect, always kept in the correct order except for one entry: the one on top or the one for the job that had the latest clock interrupt.

The infinitesimal-time-slice algorithm

Both the fixed-priority and the relative urgency algorithms occasionally perform unnecessary program switching; in the first case, as was demonstrated earlier, such switching can result in "local lockouts," while in the latter case the penalty is merely an increase in the overhead. Program switching contributes to system overhead mainly because, prior to exchanging programs, information must be saved to enable the operating system to restart the interrupted program at a later time. Although in some computers program switching is accomplished quite rapidly,* unnecessary switching should clearly be avoided. It is interesting, however, to examine another type of scheduling algorithm in which the number of program switchings is intentionally made very large. While this algorithm cannot be implemented, it provides an added insight into the CPU allocation problem and it does form a basis on which practical — though necessarily limited — scheduling mechanisms can be built.

Under the assumption that program switching requires no time at all, it is possible to consider a simple scheduling algorithm that not only guarantees timely service to all jobs, but also does not require priority determination and, in addition, establishes a clear connection between the various PCL's and the instantaneous CPU loading. The essence of this algorithm — the "infinite time-slicing" algorithm — is the division of the time axis into very small periods Δt; in each such period, the CPU is allocated, in some arbitrary order, to all active jobs such that each job receives $\alpha_i \Delta t$ CPU time, where α_i is the PCL for job i. If Δt is infinitesimally small, then regardless of the relative orientation and length of the clock intervals, each job accumulates exactly T_c CPU time ($\alpha_i T_t$) in every clock period T_t. Moreover, the condition that the sum of all PCL's is no greater than 100% is then sufficient to guarantee that there will be enough CPU time to satisfy all jobs.

An approximation to the infinitesimal-time-slicing algorithm can be effectively realized on computers that provide extremely fast program switching. An external "commutator-clock" hardware is set by the operating system to generate interrupts that divide a basic period Δt into sub-segments. Each subsegment is assigned to a given program. Program switching at the interrupt times is unconditional, and no program has access to the CPU except during its assigned "slot" of the commutator cycle. With a basic period Δt of, say, 200 microseconds, and program switching time of 3 microseconds, five time-critical jobs can share the CPU at the cost of 7.5% overhead. Since no job can "overflow" in its CPU usage, system integrity from that point of view is guaranteed. A "slop" equal to the period Δt must be allowed for by each user.

The major disadvantage of this scheme is that it is inflexible and becomes highly inefficient in the presence of asynchronous interrupts and certain more complicated types of hybrid jobs. These are discussed next.

Multiple complex jobs

The simple model of a hybrid job that has been assumed up to this point is not always adequate. There are two important cases in which a more complex representation is needed. One such case occurs when the job has, in effect, two clock periods:[4] a short one (say 10 milliseconds) for computations involving high-frequency signals, and a long one (say 60-100 milliseconds) during which slowly-changing variables have to be updated. In general, this type of job is accommodated by setting its clock interval to the shorter of the two periods; then on every clock the "fast loop" computation is performed first followed by a segment of the less urgent task. This arrangement usually results in unequal compute times (T_c). The second case is the one in which one or more of the time-critical jobs insist on defining private, multi-level priority interrupt schemes to handle "unpredictable" conditions; this requirement arises, for example, in one possible imple-

**Of course the sort increases the monitor "blind spot."

*The CDC 6000 series computers, for instance, exchange two programs in less than three microseconds; this includes saving and restoring 24 operating registers and other information.

mentation of hybrid function generation. The question is whether the CPU can still be efficiently scheduled when two or more of the time-critical tasks are of the more complex types described above.

Perhaps it's best to answer this question with an illustration. Let job A have a clock period T; it requires a .75T and .25T compute time alternately. Its average PCL is 50%. If the relative urgency algorithm is used, so that "local lockouts" need not be considered, then job B may not demand more than 25% PCL if its clock period is T, but it can have up to 50% if its clock period is 2T. The conclusion is that the sum of the PCL's being no greater than 100% is no longer sufficient to guarantee unconditional CPU scheduling when one or more jobs have unequal CPU requirements in several clock periods. It is now necessary to compare the clock periods of the various jobs and their CPU requirements during those periods; furthermore, this comparison must be done either on a worst-case basis (which, in many cases, is not obvious), or by considering all possible situations. The latter approach can become exceedingly difficult, particularly when there are several time-critical jobs in the system, with one or more having variable compute time and/or variable clock periods. It is also clear that this analysis, which can require lengthy arithmetic computations, must be done prior to admitting the next time-critical job into the active list. The redeeming feature is that the analysis itself is not time-critical; it can be done relatively at leisure. A special system program—the "static scheduler"—is initiated by the monitor or some other component of the operating system, whenever a new time-critical job seeks entry into the system. This system program can be handled just like a batch job, except that its priority must be high enough to avoid undue delays in loading new jobs, and it must have means of communicating with the operating system. That is, it receives data describing the current time-critical load of the system and the three parameters (T_t, T_r, T_c) of the new job and it outputs a go/no-go indication based on its analysis of the situation.

The handling of asynchronous interrupts

Users of hybrid computers have often tended to put a high value on the availability of multi-level priority interrupt hardware. Our own view is that such hardware is highly over-rated; we know of no instance where multi-level priority interrupts were of a decided advantage in a hybrid application, except to facilitate background batch processing with one foreground hybrid job. Nevertheless, hybrid users will undoubtedly continue to insist on this feature (for emotional security if for no other reason). In a multiprogramming environment, the entire concept of multi-level interrupts private to a given job needs to be thoroughly re-examined. Such examination brings to light several aspects of interrupts in general that are often overlooked even in uniprogrammed situations.

An "interrupt" is a signal that signifies the existence of an external condition that requires some action on the part of the digital computer. In particular, the interrupts that are of interest to the CPU scheduling mechanisms are those that require CPU activity. Such interrupts are characterized, explicitly or implicitly, by the same three parameters that describe any other hybrid job; repetition period, tolerance, and compute time. It should be obvious that, if no limit can be put on the maximum frequency of repetition, then a situation could easily arise in which even a CPU that is entirely committed to the servicing of these interrupts cannot meet the demand; that is, it cannot complete the computation of one or more interrupts within their tolerance. Thus truly "asynchronous" interrupts—that is, completely unpredictable ones—are not practical in any case; some information regarding their requirements must be available, or, at least, estimated. However, when the CPU is committed to one user, he can find out, by cut-and-dry methods, what is the frequency of occurrence of his interrupts. In a multiprogramming environment the user must define these parameters to enable the static and dynamic schedulers to determine whether and when his interrupts can be serviced.*

The crucial question in regard to private interrupt schemes in a multiprogrammed system is that of their relative priorities. If an interrupt of job A occurs while job B is in possession of the CPU, should the control over the CPU be given to the requesting job or should it remain with the one in execution?

The simplest answer is to define a nominal "job priority" hierarchy such that job A has a higher absolute priority than job B, and so forth. The penalty is that job B's interrupts cannot be processed until job A, and all other higher priority jobs, release the CPU. This scheme is unsatisfactory not only because it limits the allowable tolerance of lower-priority jobs, but also because it is unworkable in the sense that if one job's interrupts are permitted to override other jobs' normal (periodic) execution, then one job can cause another to "lose synchronism" (that is, not finish its assigned task by the next clock interrupt) simply by generating enough interrupts. The only absolutely safe way to operate under these conditions is to limit the interrupts of any job so that they are allowed to interrupt

*The situation is similar to that of the analog programmer who is required to estimate maximum values for his problem variables for scaling purposes.

only when the CPU is in possession of the respective "parent" job; the user simply allows for interrupt processing by declaring a larger PCL. The effect is that the interrupt tolerance of any one job is limited to no less than that job's clock period. In many cases this is not a serious limitation, although it can lead to rejection of jobs that declare short clock periods (due to the need to handle interrupts) even though the system is otherwise lightly loaded. In most hybrid tasks fast interrupts are (or can be made) periodic since they are either under the programmer's control or generated by hardware, which is inherently periodic or nearly so; those that cannot be made periodic are usually due to manual action and are, therefore, relatively slow—they can be scanned for activity on a periodic basis.

The nature of the difficulty in scheduling interrupts is the same as that of scheduling periodic jobs: it is not possible to allocate the CPU at full efficiency, and at the same time guarantee protection to all users if the priority levels are static—that is, predetermined by hardware or fixed at load time. This is why priority interrupt hardware in its present form is not useful in hybrid multiprogramming. One way in which asynchronous interrupts can be handled is through dynamic allocation of priorities based on least-time-to-go criterion. It is in the scheduling of asynchronous interrupts that the relative urgency algorithm finds its strongest justification. The algorithm simply regards interrupt subroutines as independent jobs from the scheduling standpoint. In determining priorities the scheduler uses the tolerance, rather than the repetition period, of the interrupt routines. Repetition periods estimated by the programmer must be for worst-case conditions. The appropriate linkage mechanisms to connect any number of interrupt subroutines to hardware signals on one side and to a single parent job on the other must, of course, be provided.

Static scheduling and facility management

The task of protecting the system against the entrance of jobs whose stated requirements exceed the available resources is entrusted to the static scheduler program. (Of course, the system also needs protection in case a job exceeds its stated requirements after being admitted; this is discussed later.) The complexity of the static scheduler program depends on the limitations imposed on the time-critical jobs. In a system designed to handle only two such jobs, with fixed clock periods, tolerances, and compute times, the static scheduler need only verify that the sum of the declared PCL's for the job in execution and the one that seeks entry to the system is no greater than a fixed limit. That limit is 100% if the relative urgency scheduling is used, or 83% when priorities are fixed by the relative length of the clock periods; of course these figures must be reduced to allow for system overhead. When the system handles several time-critical jobs, with varying parameters and with associated asynchronous interrupts, static scheduling is more involved. One approach to the problem is to have the static scheduler effectively *simulate* the operation of the system over a sufficient length of time. Since, in general, the worst case occurs when all jobs, periodic and interrupts, require service simultaneously, the simulation can begin with this situation and extend through the lowest common multiple of the repetition periods (asynchronous interrupts must declare worst-case repetition rates). Of course, unless severe limitations are imposed on the resolution of the clocks and on the number of jobs, this lowest common multiple can be a very large number. Assuming it is not, the static scheduler then constructs a "schedule-of-events" in which, by using the declared repetition periods, all future requests for CPU service are listed. Using the declared compute times and tolerances, and assuming the correct CPU scheduling mechanism, the scheduler can step through this sequence of events and quickly detect cases where a job fails to get its share of the CPU time. When this happens, the scheduler returns a "no-go" message to the operating system, which then rejects the job seeking entrance and issues appropriate message to the operator or to the remote station involved.

Scheduling algorithms of the infinitesimal-time-slicing type greatly simplify the static scheduling problem. This is a significant advantage since the satisfactory solution of this problem in the most general case—that of a job mix with complicated, varied parameters and asynchronous interrupts—is very difficult. The penalty paid in avoiding the issue is in inefficient servicing of jobs with unequal response and repetition periods (interrupt routines). Consider, for example, a "standard" job with $T_c = 85$ milliseconds and $T_r = T_f = 100$ milliseconds, operating with a second job with $T_c = 200$ microseconds, $T_r = 1$ millisecond and $T_f \leq 4/3$ milliseconds. Relative urgency scheduling can satisfy both tasks (assuming no overhead or delays) because the effective PCL of the interrupt routine is $200/1333 = 15\%$, and that of the standard job is 85%. In the infinitesimal-time-slicing case, however, the interrupt routime must claim 20% PCL (200 microseconds in every millisecond) so that the two tasks cannot be accommodated.

In many practical situations static scheduling is not complicated, because the number of different job combinations that are possible is not large (most time-critical hybrid jobs are long term ones) and is limited by factors other than CPU scheduling.

When a job is rejected on grounds of insufficient resources, two questions arise: (1) How does such a situation come about? and (2) How can it be prevented? The answer to the first question can easily be visualized in terms of the history of the installation. Large-scale installations almost never "happen"—they tend to grow from relatively modest beginnings. The central digital computer, with a modest amount of core and one linkage system, are probably the first elements to become operational. Time-critical users at this stage must alternate on the system, so that when one is on it, he has all its facilities at his disposal. Unless firm restrictions are imposed at this early phase, users will tend to be liberal in core and processor usage. When the subsequent linkage systems are installed and new jobs begin to appear, the "veteran" jobs often find that they cannot get on the system in the presence of the new jobs. A general belt-tightening then takes place, more core ordered, and so forth.

When the system operates in a "closed-shop" mode, such conflicts can usually be ironed out fairly easily. Resource allocation becomes a crucial problem when the system is used by remote and/or politically powerful "customers," each developing several hybrid jobs of different magnitudes. The only way to guarantee that conflicts do not arise is to impose in advance core and CPU limits on each customer. The static scheduler can check the parameters of any incoming job against these limits (they should be tamper-proof system parameters, but nevertheless relatively easy to modify) and reject the job when it exceeds these limits, even though enough resources are available at the time.

This procedure may be unpalatable, since it requires close coordination (in many cases amounting to negotiation). But note that it is clearly superior to the option of installing separate, committed systems, in which the resource allocation decision is made once and for all with few means of modification. Resource allocation is a continual management responsibility; the multiprogrammed hybrid system allows management to manipulate the allocation dynamically in response to varying corporate priorities.

The master-schedule approach to CPU scheduling

An interesting system configuration in which users are *not* permitted to have individual clocks and which eliminates the need for dynamic CPU scheduling deserves a mention. In this system the single central clock is under the sole control of the operating system. The clock is used as a "next event indicator": every time the clock count is exhausted, the system loads into it a count representing the period from the present instant to the next event of significance. The "events" represent the various phases of the different jobs (such as "clock interrupt," end-of-compute-time, end-of-tolerance, etc.). These events are stored in a "master schedule" in a manner similar to that of the static scheduling program already described. Thus CPU scheduling is totally static and is handled on a batch basis concurrently with the running time-critical jobs.

The effective implementation of this scheme requires the clock periods of the various jobs to be related in one way or another; for example all periods may be required to be integer multiples of some period Δt. Suitable algorithms are of the infinitesimal-time-slice type. They suffer from a number of disadvantages. The period Δt is often dependent on the particular job mix and thus is variable from day to day or even during the day. In some cases such variations are intolerable, since the *accuracy* of the clock is often much more important than its *resolution*. A job may be satisfied with a resolution of a millisecond (defining its clock interval to the nearest millisecond), but once a period is selected the job may require that this period remain unchanged within a microsecond. Another disadvantage, as discussed earlier, is that asynchronous interrupts cannot be serviced as quickly as is possible with other scheduling schemes.

Compressed-time computation

Much of the recent effort in hybrid computation has been in the area of compressed-time operation (also referred to as high-speed repetitive-operation, HSRO, or faster-than-real-time operation). Compressed-time computation places heavy demands on the analog computing components and has some implications regarding the organization and design of the linkage system. As far as the digital computer is concerned, however, the difference between slow ("real time") and compressed-time operation is a matter of scale rather than of a fundamental change. Figure 1 still describes adequately the situation, although sometimes a number of $A \leftrightarrow D$ transfers take place prior to the digital computation and, of course, the analog computer cycles between OPERATE and IC for every digital compute step. The repetition period T_r is now considerably shorter than in the "real-time" case, so that monitor delays play a more prominent role in determining the permissible useful load of the system. These delays, resulting from the "blind spots" in the monitor loop, can, in a multiprocessor system, be minimized by relegating the more time consuming monitor tasks to other components of the operating system. In a uniprocessor system, hardware can perform some monitor tasks (e.g., scheduling and accounting).

Related considerations

The case for multiprocessing

Up to this point the discussion of hybrid multiprogramming did not specifically consider multiprocessing features. It was tacitly assumed that the CPU is sufficiently fast and that the core size is large enough to serve the simultaneous time-critical tasks; it was also assumed that the hardware and software components of the system are inherently multiprogramming-oriented, although some special modifications and additions may be needed (static and dynamic CPU schedulers, for instance) to accommodate the time-critical requirements. Several third-generation computers that offer these features (fast CPU, large core, multiprogramming) also offer multiprocessing hardware, and it is interesting to examine how such hardware can be used in the context of hybrid operation.

At present there appear to be two significant trends in multiprocessing. In one case the concepts of, on the one hand, independent, self-governing I/O channels, operating, to a large extent, concurrently with the compute hardware, and, on the other, of satellite processors intended to relieve the CPU from I/O chores, led to a development of independent "peripheral processors," each with a repertoire of logical and simple arithmetic instructions, and sometimes, its own private memory. The CPU, which is very fast, performs computation only, while the peripheral processors handle I/O, and I/O related operations, simultaneously with the CPU.* The advantage of the peripheral processors is that the CPU is never burdened with I/O control or system functions. All I/O, including A-D and D-A channels, is handled concurrently, without any CPU intervention, by peripheral processor programs. Moreover, the monitor, and other components of the operating system, reside in peripheral processors and execute in parallel with each other and with the CPU. System overhead accumulates only when a job is done and wishes to release the CPU, but the monitor does not honor this request immediately due to other tasks that is is performing at the time.

Another approach is to add more CPU's, all identical and all allocatable to users and to the operating system.** With overlapped, interleaved memories of sufficient size such additions can represent very nearly doubling, tripling, etc., of the computing power.
PCL's for which is greater than 100%. According to Grosch's Law, one ought to be able to get for the cost of two processors a single processor that is four times as fast, so that multiple CPU's may not be attractive from the economic standpoint. Moreover, it is easy to show that a single, faster CPU is more desirable for handling time-critical tasks than a number of lower-speed units, at least in some cases. Consider two systems, one with a single CPU and the other with two CPU's half as fast. Let there be three time-critical jobs, with identical clock periods, each requiring 33.3% PCL on the faster unit. If all three jobs require service simultaneously, the fast CPU can provide service to all three, but the two slower units will require 66.7% of the clock period to service two jobs, and the remaining time will not be sufficient to serve the third.

*The CDC 6000 series represents such an architecture.
**As in the IBM System/360 Model 67. The Burroughs B8500 permits multiple central and peripheral processors.

As far as the time-critical jobs are concerned, the presence of multiple, identical central processors simply allows the scheduling of a job mix the sum of the

Core allocation

In the present state of computer technology, time-critical hybrid jobs must be immune to core relocation and swapping procedures that are common in time-sharing and multiprogramming systems. The need for such relocation arises in two different ways. In systems using paging techniques, missing pages must be fetched into working storage, evicting other pages that are already in core. In systems without paging, relocation takes place to consolidate the scattered core segments that are in use and provide contiguous working storage for an incoming job. These techniques cannot be freely used on the time-critical jobs since the time to access mass storage devices, or to relocate a sizable segment of core, is of the same order of magnitude as the frame time (clock period) of these jobs.

A system with hardware paging in which the construction of physical addresses does not slow down the computer to any significant extent and in which the time-critical jobs can be loaded *in toto,* and then remain immune to page turning procedures by not relinquishing their assigned pages, appears to be ideal provided the pages are small enough so that the amount of unusable core (remainder of the last page) is minimized. In the absence of paging, a core-allocation algorithm is needed to minimize gaps and the size of core relocations when required. If only two time-critical jobs are expected to co-exist, the algorithm is quite simple: the jobs are located at the opposite extremes of core. The effect is simply that, from the standpoint of the relocatable jobs, the contiguous core size is reduced by the amount taken by the time-critical jobs. When more than two time-critical jobs are possible, the system attempts to push these jobs alternately against the opposite extremes of core. As jobs terminate, gaps may develop. The core allocation algorithm attempts to fill these gaps with the minimum of relocation. Note that

there are periods during which hybrid jobs are *not* time-critical, for example during initialization, post-run editing, and certain phases of debugging.

Protection and independence

A hybrid multiprogramming system requires protection and independence for its jobs and operating system similar to those that are required in any multiprogramming environment.[6] For example, user jobs are not allowed access to areas of core that are not assigned to them; these jobs are not permitted to execute certain I/O instructions, etc. Beyond these elementary precautions are added requirements due to the presence of time-critical jobs. For example, the classical solution to the problem of preventing a user's job from "hogging" the system is to "time-slice." Each job in working storage periodically receives control of the CPU until one of three things happen: (a) the job issues an I/O request; (b) the "time-scale" is exhausted; (c) the accumulated execution time of the job exceeds the time limit declared on the job card. In the latter instance the job is evicted from the system.

With time-critical jobs *any* excess in CPU usage by any one job is intolerable, since such excesses can cause other jobs to "miss" their next clock. Hence a more or less continuous surveillance mechanism must be set up to monitor each time-critical job execution. The compute time declared by the job is entered into a table at the beginning of every repetition period, and the CPU time accumulated by the job is subtracted from this figure. If the figure reaches zero before the job signals the monitor that it has completed its task, the job is removed from the time-critical list and a "lost synchronization" error message is sent to the job. Depending on the sophistication of the system, the job can then be given a chance to find out what happened by reducing it to batch status and thus giving it control of the CPU at a low priority level. The job can then execute a post-mortem routine or go into an editing phase, formatting and outputting the data accumulated in mass storage during the run up to that point. A less kindly system simply evicts the offending job. Note that an interrupt routine must be timed as an independent job with its own repetition and compute time parameters.

The independence of jobs is also slightly more involved in the time-critical case. In general, hybrid jobs begin execution in a non-time-critical phase, during which both analog and digital initial conditions are set, linkage instructions issued, and possibly, automatic analog set up is performed. These tasks may involve many accesses to mass-storage and other time consuming tasks. At the end of the run the job may execute complex, lengthy data reduction and editing subprograms. If hybrid priorities are assigned on the basis of clock intervals, then a high priority job loaded in while a low priority job is already running can "lock out" the latter for a long period while the high priority job is initializing. A similar situation occurs when a high priority job goes into the editing phase while low priority jobs are still executing. It is, therefore, necessary to provide a mechanism for switching priorities from low at load time to high at the end of initialization and back to low at the beginning of the editing phase. When the relative urgency algorithm is used this is accomplished simply by inserting very large time-to-go figures in the stack when the job is in initialization or editing. of course, the standard means for assuring independence in any multiprogramming system—relocatable object code, logical identity of I/O units, etc.—must also be present in the hybrid case.

A multiprocessor system, particularly one that has many lower level processors, has a distinct advantage here in that the necessary mechanisms for assuring protection and independence can be performed simultaneously with CPU computation.

In some computers, such as the Burroughs B8500, Univac 1108, the software and hardware are designed to produce and handle "re-entrant" code. The assembler or compiler generates object code that is segmented into pure procedure sections and data sections. The hardware includes separate base and limit registers and options for read protect, write protect or both for each segment. These features permit better utilization of working storage without compromising the inter-user protection level. Thus several users can share a single procedure segment that is in a read only mode with respect to all users while maintaining private data areas that are both read and write protected with respect to all users but the owner. Significant examples of the use of this feature are in the re-entrant compiler and the I/O routines that serve many users simultaneously.

The role of the remote console

The "committed computer" appearance that the hybrid multiprogramming system attempts to present to each user is not complete unless the user, at a remote location, has a significant measure of control over and monitoring of his job through the various stages of checkout, validation and production. The need for such remote control is particularly great when remote analog computers form an important part of the hybrid configuration, since the analog input and output of the simulated system is of immediate concern and many (if not all) parameters of interest are part of the analog

setup. These control and monitoring functions, exercised at a "remote terminal" (typically consisting of a CRT, a keyboard, and a low-cost printer), include symbolic editing and display of the source program, tracing and display of the object code, mode control of the entire hybrid simulation and so forth.

To permit this sort of remote control the system must be endowed with certain important software components. There should be a fairly sophisticated file-managing capability. The user's source program can be maintained in mass-storage, while the user is allowed to edit it by the insertion and deletion of statements, including control statements. Preferably there should be two levels of editing: temporary and permanent. The temporary insertions and deletions can be used to "try things out" while the program is in checkout. All such temporary editing of the source file can be nulled on a command from the remote console. After editing, the user can request a compilation, and receive at least a "go/no-go" reply indicating a successful compilation or the presence of diagnostics, respectively. In the former case, the user can then submit the resulting binary file to the input queue for execution.

During execution the user should be able to display his job's status (executing, idle, waiting, etc.); interrogate any memory location using relative addressing (that is, the job's first location is address zero regardless of its physical address) in any of several convenient formats (say octal, character, decimal integer and decimal floating point); modify any memory location in any format; display and modify the contents of all operating registers at the most recent instant when the program ceased execution; breakpoint through his program; and so forth.

At this writing (March 1967) many of these capabilities are in daily operation, and almost all others are in the process of implementation.

Mass storage scheduling

In the early days of hybrid computation little use was made of mass storage devices. Now these devices are widely available for use even with very small computers. They can be effectively employed in function generation, in accumulating run history or data generated by simulation hardware (e.g., telemetry), for maintaining initial conditions files and component availability files (for analog static check software) and so forth. Some of these uses are clearly not time-critical at all (e.g., static check), while others vary in the degree of their urgency. For example, failure to bring the next function table to working storage in time is almost certain to be fatal to a function generation program, whereas the loss of one frame of telemetry or run history data may often be overlooked. The question, then, is whether and how to schedule the operation of mass stores to guarantee service to time-critical tasks.

The situation with regard to the scheduling of mass storage devices is fundamentally different from that of the central processor, because the degree of urgency of the services they are required to provide can be controlled by manipulating the size of the relevant working-storage buffers. Thus, if a function generation program is allotted twice as much storage for function tables, the mean time between fetches of new tables from mass-storage is doubled. Hence the major concern is the maximization of throughput by minimization of mass-store idle time.

A recent study[8] found that a dramatic improvement in the utilization of fixed-head devices occurs when the users are serviced not on the basis of least-time-to-go or first-come-first-served, but with a shortest-access-time first policy. Rather than honor the CPU priorities, this policy establishes its own priority hierarchy in which the job that requests the information that is currently available under the read/write head has the highest priority. For moving-head devices, a uniform scan policy is suggested. The arms move across the tracks uniformly back and forth, and while the heads are on any given track, the shortest-access-time-first scheme is followed. While this scheme may not be justified with a disk for which head positioning time is not a function of distance moved, it is nevertheless an interesting proposition.

Linkage considerations

The design of the analog-digital linkage system for a hybrid computer is a subject that deserves much more attention than that which can be given it in the present context. The object of this section is, therefore, to briefly discuss a few aspects of the problem that are particularly related to the multiprogramming environment.

In a hybrid multiprogramming system a basic difference exists in the mode of operation of the digital and analog computers: the former is shared by many programs while the latter are committed, individually or in groups, to specific programs. The linkage may be operated in either mode.

For example, each analog console may be assigned its own linkage system, consisting of AD/DA channels, a clock (or clocks) and control logic. The main advantage of this scheme is that it permits one-time set-up (patching) of the analog program, which need not be changed unless, perhaps, the program is assigned to a different set of consoles (it is rarely, if ever, possible

to guarantee that all consoles are exactly alike). One disadvantage is that those linkage components in a given console that are not used by the program cannot be assigned to another job and hence must remain idle. In addition, if the job is reassigned to different analog consoles, the digital program must be modified to request different physical AD and DA channels.

Another possibility is to construct a pool of linkage elements (AD/DA channels,* clocks, control elements) that can be assembled in an arbitrary fashion, through commands from the digital computer, into appropriate groups to serve specific jobs as they enter the system. The advantages of the pool approach are that the available resources can be allocated dynamically, the number of unusable elements is reduced to a minimum, and that linkage elements can be referred to by the programs as logical, rather than physical units. However, it is not known in advance what physical elements will be assigned to which program; in fact the assignment can change from day to day depending on the number of jobs and the order of their loading. Hence it is necessary that the outputs of all DA channels and the inputs to all AD channels terminate simultaneously on all analog consoles. This, in turn, requires that adequate protection is present in the digital computer, to prevent one job from accessing channels (and more importantly, control elements) assigned to another job; and in the analog consoles, to prevent interference when one job accidentally signals to a channel assigned to another job. Of course, as the system load varies, individual jobs starting operation may be required to repatch the AD/DA partion of their setup.

A third configuration is possible which strikes a compromise between these two extremes. In this case linkage elements are preassembled into a number of independent "links." These links contain all the necessary elements to provide service for one given job, but they are not associated with any particular analog console. In order to connect these links to a variable grouping pattern of analog consoles, an "interconnect box" is provided in which all AD and DA channels terminate, as well as groups of uncommitted trunks from the analog consoles. As the load on the system changes, changes in the patching of the "interconnect box" may be required.

In several respects this last configuration combines the worst features of the other two. Errors in patching the "interconnect box" for an incoming job can affect jobs already running. Channels must be referred to as physical units, and unused elements must stand idle. Unfortunately this scheme is, currently, the easiest to implement.

One item which may be worth including in the linkage section is a hardware equivalent of the dynamic scheduling algorithm. The information needed for determining relative priorities according to the least-time-to-go principle is already available in the various clocks. The hardware for making the necessary comparisons is neither complex nor expensive. The hardware scheduler can inform the monitor which time-critical job should receive control of the CPU, thus relieving the monitor of considerable bookkeeping.

The clocks, incidentally, may very well be arranged in pairs to enable the links (and the scheduler) to time not only the repetition periods but also tolerances.

One requirement that appears to be rapidly developing is for inexpensive links consisting mainly of low precision DA channels or incremental plotter drivers. Such elementary linkages are useful (in conjunction with CRT-keyboard terminals) when the digital computer performs essentially analog jobs, perhaps using MIMIC or a similar simulation language. The speed of the central computer enables it to offer cost-per-computation competitive with analog costs, as well as on-line plotting of the results at a precision equal to or better than that of the analog and with a dramatic improvement in repeatability and ease of set-up.

CONCLUSION

Our primary motivation in writing this paper has been to elicit reactions and comments from present and potential users and designers of hybrid multiprogramming systems. The economics of multiprogramming, the handling of asynchronous interrupts, resource allocation algorithms, linkage system configuration and many other subjects need to be examined in more detail. The analog-hybrid department of McDonnell Automation Company is currently servicing on a routine basis two time-critical hybrid jobs simultaneously with batch processing. In addition, simulation work—both analog and digital—is now in progress at the Automation Company in an attempt to verify the validity and efficiency of various CPU schedulers under a variety of load conditions. However, this particular area can benefit from additional theoretical work. It may prove interesting and useful to try to adapt queuing theory to the hybrid CPU scheduling problem, where the duration and amount of service and even arrival times are predictable but the permissible waiting time is rigidly constrained. All in all, the field appears ripe for many interesting investigations.

*No distinction has been made between AD/DA data and discrete channels since they can be made, by proper design, to appear logically identical to the digital computer. Sample/hold circuits also have not been specifically mentioned since they are assumed to be part of the AD channel circuitry.

ACKNOWLEDGMENT

Most of the ideas presented here evolved (and are still taking shape) in an environment of informal, continual debate within the analog-hybrid section of McDonnell Automation Company. It is, therefore, difficult to define individual contributions. John Clancy, Luke Abkemeier, Frank Brown, and Jerry Lewis have actively participated in the debate, and Don Augustin is responsible for injecting a measure of practicality at many points where it bordered on fantasy. Members of the Communications and Special Systems Division of Control Data Corporation made, indirectly, certain contributions.

REFERENCES

1 G H GALE
The Boeing Huntsville simulation center
Simulation vol. 5, no. 4 October 1965

2 R BELLUARDO R GOCHT G PAQUETTE
A time-shared hybrid simulation facility
Proceedings of the Spring Joint Computer Conference 1966

3 D C AUGUSTIN J C CLANCY M S FINEBERG
Hybrid computation with a large scale multiprogrammed multi-processor system
Unpublished, available from the authors McDonnell Automation Company

4 E L MITCHELL J B MAWSON J BULGER
A generalized hybrid simulation for an aerospace vehicle
IEEE Trans. Elect. Comp. vol. EC-15, no. 3 June 1966

5 J J CLANCY M S FINEBERG
Hybrid computing: a user's view
Simulation vol. 5, no. 2 August 1965

6 A J CRITCHLOW
Generalized multiprocessing and multiprogramming systems
Proceedings of the Fall Joint Computer Conference 1963

7 M S FINEBERG
Letter to the editor
Simulation vol. 9, no. 2 August 1967

8 P J DENNING
Effects of scheduling on file memory operations
Proceedings of the Spring Joint Computer Conference 1967

The IADIC: a hybrid computing element

by JAMES I. CRAWFORD and MORRIS J. BODOIA
Martin Marietta Corporation
Orlando, Florida

INTRODUCTION

One of the major problems encountered in a hybrid computing facility is interfacing the analog and digital copartners. The papers of Hagan and Treiber,[1] Rubin,[2] and Chapelle[3] demonstrate the versatility of the MDAC (Multiplying Digital to Analog Converter) as an interface element.

The MDAC is a hybrid computing element utilizing analog and digital inputs to produce an analog output. The purpose of this paper is to introduce a hybrid analog to digital converter which the authors have named "IADIC" for "Integrating Analog to Digital Converter." The IADIC concept was introduced by Jarret[4] in 1960 but has apparently not received adequate attention. As the name implies, the IADIC integrates an analog voltage and produces a digital output. Integration is performed by a conventional analog integrator with its inherent advantages of speed and continuity. The integrated output is accumulated digitally thereby eliminating the resolution problem inherent in analog computation. To summarize, the IADIC can provide real time, continuous integration with digital precision. These characteristics are very desirable in hybrid guidance and control simulations where high frequency control variables are usually integrated in the analog domain and converted to digital representation for high precision guidance and trajectory calculations.

In conclusion, the error analyses of Jarret[4] and section IV of this paper indicate that as an integrator the IADIC accuracy should be at least as good as the better analog integrators.

I. Theory of operation

A. Description

An IADIC has been mechanized in Figure 1 by combining an analog integrator, comparators, logic, switches, and an up-down counter. When this IADIC is in the operate mode, the scaled integral of the input $\left(\frac{X(t)}{q}\right)$ will be generated as an output and will be represented as a digital count in an up-down counter. When the IADIC (see Figure 1) is in the operate mode, the analog input $-\dot{X}(t)$ is integrated until the integral exceeds either the plus or minus comparator (quantum) level. When this occurs, the logic element gates one of the constant width clock pulses into one of the normally open switches S_1 or S_2. By closing the proper switch for an accurately-known small time, a square wave feedback pulse of opposite polarity to $-\dot{X}(t)$ is generated. The amplitude and width of the feedback pulses are adjusted so that the integral of one pulse is exactly equal to one quantum. Thus the effect of a feedback pulse is to return the integrator output to zero. Simultaneously with the feedback pulse, a count pulse is sent to the up-down counter. An up or down count is sent, depending on which comparator has been violated. The final step in the IADIC operation is to multiply the counter content by the quantum value "q," and the result of this multiplication is a digital number representing $X(t)$, the integral of the input. This multiplication is a scaling operation performed in the digital computer.

In review, the IADIC operation may be outlined as follows:

(1) $\dot{X}$ is integrated until X changes by one quantum level.
(2) A feedback pulse, which returns the integrator to zero, is generated.
(3) A count pulse is sent to the up-down counter.
(4) The process is repeated.

Since the IADIC is an analog to digital converter, it must be considered as a quantizer. Susskind[5] defines a quantizer as a device which converts an input which

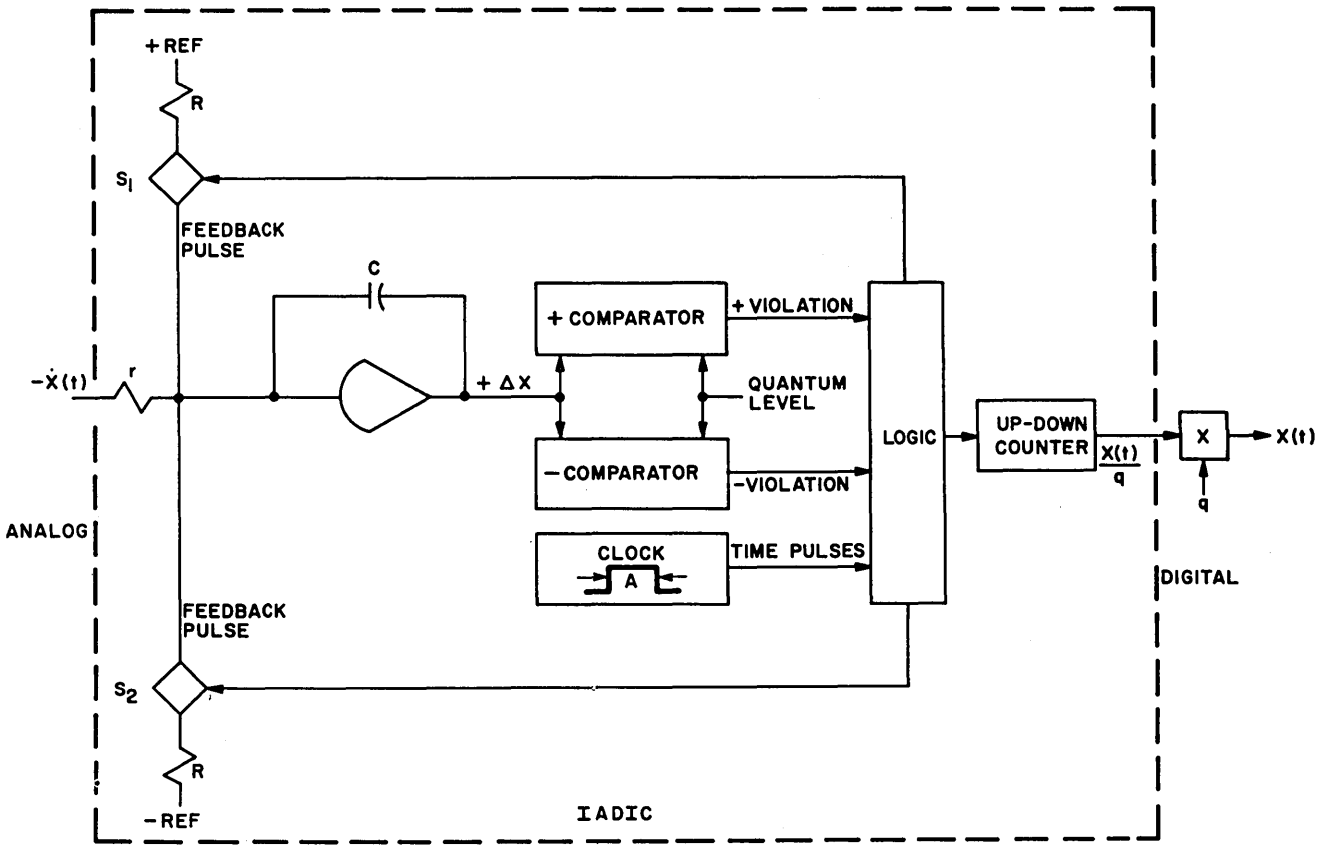


Figure 1—IADIC mechanization

is continuous into an output which has only discrete values. This quantizing action is shown in Figure 2 where q is defined as the quantum level (or quantizing level).

The analog integrator in the IADIC is scaled to ensure that this quantum, q (see Figure 3), will be represented by a reasonable voltage. The output of the IADIC for a step input is shown in Figure 3.

It can be seen by comparing Figures 2 and 3 that the quantizing actions of the IADIC and present analog to digital converters are identical. The IADIC, however, is scaled to represent the least significant bit or quantum level, q, by a reasonably large voltage.

An example will now be developed to clarify the scaling of the IADIC. Referring to Figure 4, the maximum range of $\dot{X}$ will be defined as 500 feet per second; and the quantum level, q, will be defined at 0.5 feet. $\triangle X$, the integrator output, will be scaled for 1.0 foot to allow for overshoot. Figure 1 will now be scaled as shown in Figure 4.

The pulse rate of the clock must be:

$$\text{Pulse rate} = \frac{1}{\Delta T} = \frac{\dot{X}}{q} = \frac{500 \text{ feet per second}}{0.5 \text{ foot per pulse}} = 1000 \text{ pulses per second.}$$

Therefore, the waveform for the feedback pulse will be as shown in Figure 5.

The period for the clock pulse, $\triangle T$, is equal to 0.001 second, and the ½ pulse period, $\tau = \frac{1}{2} \triangle T$, is equal to 0.0005 second. Now, the IADIC is "reset" by producing a feedback pulse which has an integral equal to a quantum level, or:

$$\int (\text{feedback pulse})\ dt = q$$

$$\frac{1}{RC} \int_{t}^{\tau+t} V\ dt = q$$

$$\text{Thus } q = \frac{V\tau}{RC}$$

If V is assumed to be reference voltage, R (see Figure 1) can now be solved from the above equation:

$$R = \frac{V\tau}{qC} = \frac{100 \text{ x } 0.0005}{0.5 \text{ x } 0.01 \text{ x } 10^{-6}} = 0.1 \text{ x } 10^{6} \text{ ohms.}$$

To reiterate the scaling procedure, the values of r, C, pulse rate, and R are determined as follows:

(1) r and C are chosen to represent q as a reasonable fraction (perhaps ½) of ΔX.

(2) Pulse rate is determined to be $\dot{X}_{max}/q$, or

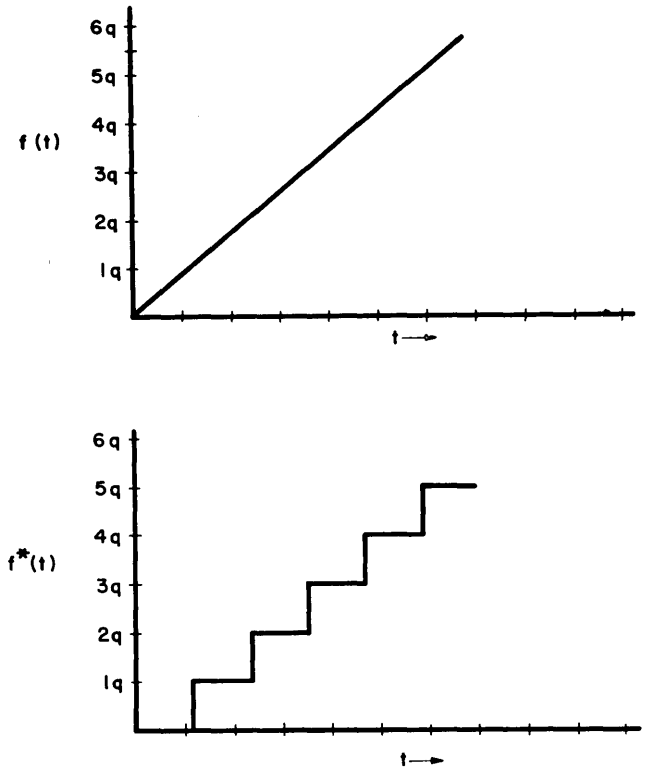


Figure 2—Standard quanitizing action of present analog to digital converters

pulse period, $\triangle T$, equals $q/\dot{X}_{max}$.

(3) R is chosen to make the integral of the feedback pulse, $\frac{V\tau}{CR}$, equal to a quantum level, q (where $\tau = \frac{1}{2}\ \Delta T$). Solving this relationship for R yields the value for R or $R = \frac{V\tau}{qC}$

B. Merits

Before discussing the error sources associated with the IADIC, some of its more obvious merits should be brought to light. These are listed below:

(1) Integration is performed in real time and continuously in the analog domain.
(2) Integration is performed in parallel.
(3) Integration is performed with digital precision (i.e., resolution).
(4) Once the integrator is placed in the operate mode, it stays in operate mode for the duration of the run. Thus no information is lost by going through repeated "Hold" and "Reset" modes.
(5) The integrator output is not limited; therefore, it can exceed the quantum level before, during, or after a feedback pulse, and again no information is lost.

II. Error analysis

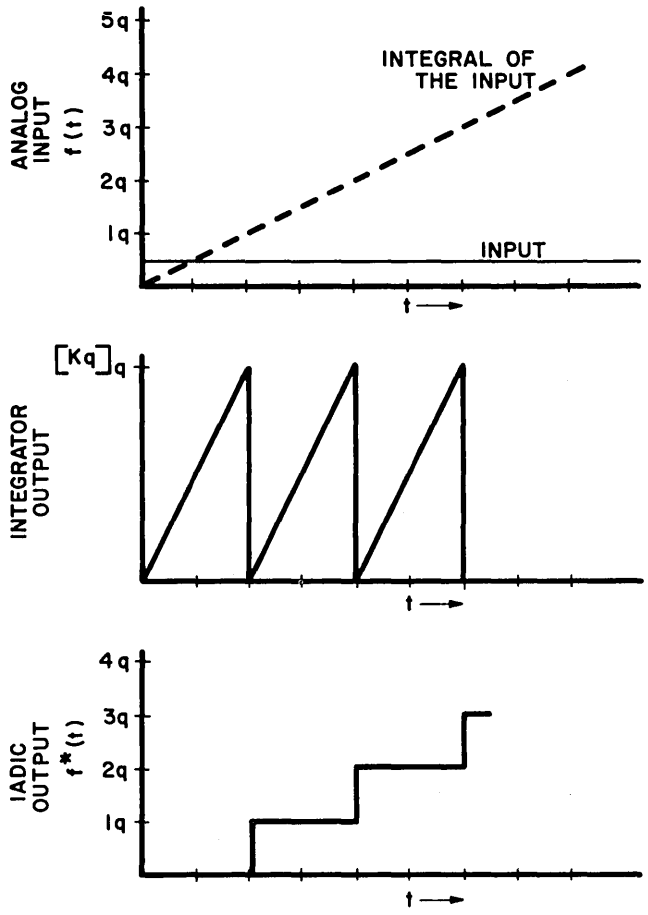


Figure 3—Quantizing action of the IADIC

The following error sources will be considered in this analysis:

(1) An error due to the inaccuracy of the comparators which sense the quantum level q.
(2) An error due to an inaccuracy in the feedback capacitor.
(3) An error in the reset pulse for the integrator.
(4) An error due to the hardware limitations.

It will be shown that the first two errors are canceling or negligible. The significant error sources are reset pulse inaccuracies and hardware limitations.

A. Comparator error

The first error to be evaluated will be comparator inaccuracy; the result of this error is illustrated in Figure 6. The quantum level at which the comparator should change state will be defined as q (see Figure 6), and e will be defined to be an error due to comparator inaccuracy. Therefore, the actual level at which the comparator changes states is q + e. The effect

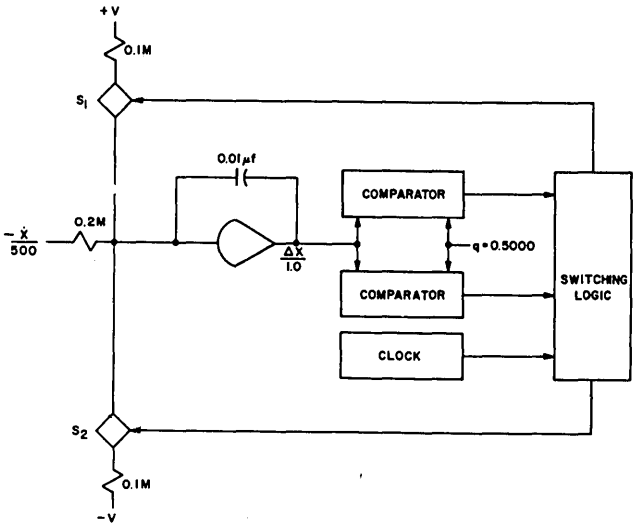


Figure 4—A scaled IADIC

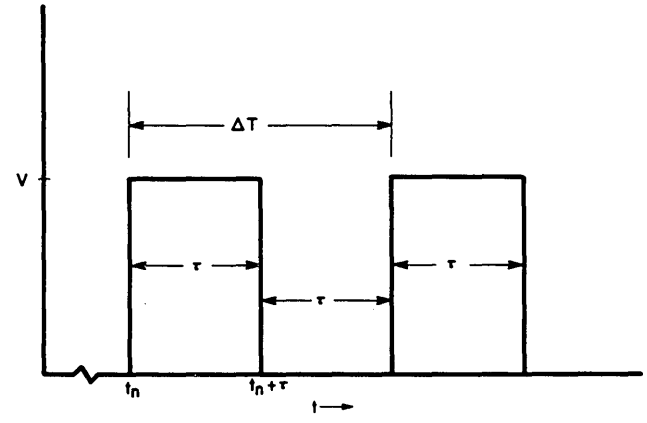


Figure 5—Typical wave shape for a clock pulse

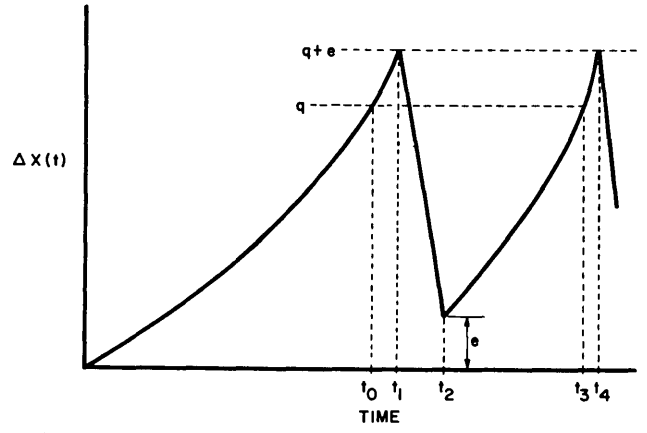


Figure 6—Effect of comparator error

of this error is to cause a feedback pulse to be generated at t_1 instead of t_0 (the proper time), but the feedback pulse will reduce X(t) by the correct value (q), and an error of e will be left on the integrator. On the next cycle (t_2 to t_3), the comparator will trip at q + e. This is a level change of q (not q + e) and produces a new output pulse which is properly interpreted as a change of one quantum (q). Therefore, the integral does contain an error due to the comparator inaccuracy; however, this error does not accumulate and represents only a small percentage of one quantum. The decision to apply a reset pulse to the integrator and not place the integrator in a reset mode condition was predicated on minimizing the effects of this error and the next error to be discussed (feedback capacitor error).

B. Feedback capacitor error

The next error source to be considered is an error in the feedback capacitor. The output of an IADIC with a feedback capacitor error of E is shown in Figure 7. At any time, t, the integrator output can be determined from Figure 7 to be:

(1) Output of integrator at t = output of integrator at t_{i-1} — integral of one reset pulse + integral of input from t_{i-1} to t.
Substituting actual parameters into equation (1) yields:

(2) $$\Delta X(t) = q - (1+E)q + (1+E)\int_{t_{i-1}}^{t} X dt.$$

Now, if we let the integrator span the quantum for this interval (i.e., $\Delta X| = q$), equation 2 becomes: t=ti

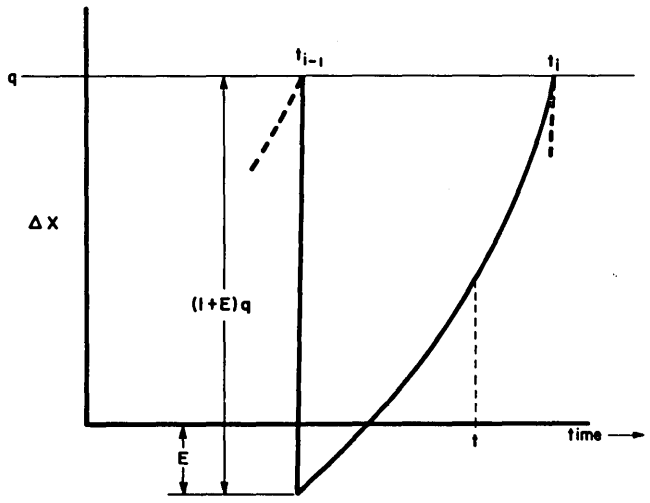


Figure 7—Effect of capacitor error

$$(3)\quad q = q - (1+E)q + (1+E)\int_{t_{i-1}}^{t_i} \dot{X}dt$$

which reduces to:

$$(4)\quad \int_{t_{i-1}}^{t_i} \dot{X}dt = q$$

Although the feedback capacitor was in error by the factor E, the actual value of the $\int_{t_{i-1}}^{t_i} \dot{X}dt$ and the output of the IADIC spanned the quantum (q) in the interval $t_{i-1} \leq t \leq t_i$. Because the input of the IADIC ($\dot{X}$) and the feedback pulse are sensed by this integrator, it is obvious that the first count of the IADIC will have an error due to E, but every subsequent count will be independent of E. The decision to apply a reset pulse to the integrator and not use a reset mode was predicated on reducing the effects of capacitor errors and comparator error.

C. Feedback pulse error

The next source of error to be considered is the accuracy of the feedback pulse. For proper operation, the integral of every pulse must be repeatable, accurately determinable, and equal to q (quantum level). A single feedback pulse as illustrated in Figure 5 will now be considered.

If the integral of this feedback pulse is defined to be q, then:

$$q_n = \int (\text{feedback pulse})\, dt = \frac{1}{RC}\int_{t_n}^{t_n+\tau} V dt = \frac{V\tau}{RC} = q$$

where

R = Feedback pulse resistor value (see Figure 1)
C = Integrator capacitor value
V = Reference voltage
τ = Pulse duration.

It has been shown that the capacitor value error does not contribute significantly to the IADIC error. Errors in the reference voltage are insignificant because all measurements are measured relative to the same reference voltage. The two most significant errors are the feedback resistor value and the pulse duration. State of the art summing resistors are adjustable to within ±0.0025 percent; however, the resistance of the switch may be on the order of 50 to 100 ohms, which is a significant error if a 100K feedback pulse resistor is used. Assuming that the feedback resistor can be trimmed to account for switch resistance, then the resistance error would be due to stability of the switch resistance during a run.

The pulse duration is a function of the "turn on" and "turn off" times of the switch. Again, if we assume that the area of the pulse can be calibrated in some fashion, then the actual "turn on" and "turn off" times are not important, but the stability of these times is important. We may now itemize the major contributors to pulse area errors by recognizing that there are three (pulse generator, logic, switch) sources of "turn on" and "turn off" time errors. These major error contributors are as follows:

(1) Switch resistance stability.
(2) Turn on time stability of the pulse generator.
(3) Turn on time stability of the logic circuits.
(4) Turn on time stability of the switch.
(5) Turn off time stability of the pulse generator.
(6) Turn off time stability of the logic circuits.
(7) Turn off time stability of the switch.

The authors were unable to obtain data on these types of errors: however, the required values to obtain reasonable accuracy can be derived. If the IADIC is to be as accurate as a good analog integrator (1.0 μf capacitor), then the feedback pulse area should be accurate to ±0.025 percent.[9] Assuming that this is a one sigma value and that all of the above error sources are independent and equal, then each of the seven error sources may contribute an error of $0.025/\sqrt{7} = 0.00945$ percent to the pulse area. Now assuming a 100K ohm feedback pulse resistor and a nominal pulse width of 10^{-4} second, the required accuracies of the seven error sources in order to obtain 0.025 percent IADIC accuracy are:

(1) Switch resistance stability = $10^5 \times 9.45 \times 10^{-5}$ = 9.45 ohms.
(2) Turn on time stability of pulse generator, logic, and switch = $10^{-4} \times 9.45 \times 10^{-5} = 9.45$ nanoseconds.
(3) Turn off time stability of pulse generator, logic, and switch = $10^{-4} \times 9.45 \times 10^{-5} = 9.45$ nanoseconds.

Although the authors were unable to obtain stability data on switch resistance, rise time, and decay time, the required values above do not seem unreasonable to the intuition when one considers nominal values of 50 to 100 ohms for switch resistances and 20 to 40 nanoseconds for rise and decay times.

It should be noted that the expected accuracy of the IADIC will be different for monotonic and nonmonotonic functions. Since independent resistors, switches,

and perhaps logic circuits are used for positive and negative pulses, the error for nonmonotonic functions will be larger by approximately $\sqrt{2}$.

To summarize, the IADIC error is almost entirely due to errors in the feedback pulse area and although little or no data are available to determine this error, a value of 0.025 percent appears to be attainable.

D. Hardware limitations

The decision to provide a resetting feedback pulse and not to reset the mode of the IADIC (see Sections A and B) when a quantum level is reached places severe specifications on the hardware of the IADIC. Because the IADIC may be in operate mode for a long period of time, integrator drift will be detrimental to the accuracy of the IADIC. Therefore, the hardware must be designed to ensure drift-free operation. Also, the hardware must be designed to obtain a feedback pulse with a repeatability which will not degrade the operation of the IADIC. Therefore, to ensure proper operation, the following hardware requirements must be met:

(1) Biases caused by any switch (especially switches S_1 and S_2 — Figure 1) in both the on and off states must be miminized and balanced.

(2) The integrator must be designed to ensure that the integrator rate limiting (due to maximum amplifier current) does not distort the feedback pulse. This rate limiting will affect accuracy and will limit the minimum feedback pulse duration. The amplifier must also be designed to minimize drift.

(3) To obtain quantum equal to a small percentage (10^{-6} percent) of the variable either the comparator levels must be set low or the integrator gain must be high (or both). As the integrator gain is increased, the probability of obtaining drift-free operation will decrease. Thus, the requirement to obtain drift-free operation will place a limitation on the maximum integrator gain, and comparator setting accuracy will place a limit on the minimum integrator gain.

(4) Switches S1 and S2 must have "turn on" and "turn off" times repeatable to 9.45 nanoseconds. To obtain this repeatability the nominal switch time should be on the order of 50 to 100 nanoseconds. Thus the current to be switched must be limited, which places a restriction on the minimum value of R and therefore the minimum quantum level.

(5) Dow,[6] Howe,[7] and Driban[8] have presented discussions on the effect of capacitor dielectric absorption on the accuracy of analog integrators. The circuit to compensate for dielectric absorption developed by Driban[8] could be incorporated into the design of the IADIC to eliminate this dielectric absorption effect.

IV. Analog to digital interface

Getting the count pulses from the IADIC into the digital computer is an infinitesimal problem compared to the normal method of analog to digital conversions which requires: (1) analog sample and hold, (2) multiplexing, and (3) analog to digital conversion. The result of this conversion is a digital number which has to be inputted to the digital computer.

Several schemes were conceived to accomplish the task of getting the count pulses from the IADIC into the digital computer. The final method for imputting the count pulses was to count and store these pulses in a special buffer register. A bank of such registers would be designed into the digital computer. This method excites the imagination because a direct line of communication is established between the analog and digital computers and emphasizes the phrase "copartners in computation."

V. CONCLUSION

Several papers[1,2,3] have been written which tend to validate the proposition that a hybrid computing element is the optimum device to interface an analog computer with a digital computer, but the authors of these papers are primarily concerned with digital to analog communication utilizing the MDAC (Multiplying Digital to Analog Converter). The hybrid computing element (IADIC) introduced in this paper should provide the flexibility and versatility to analog-to-digital conversions that the MDAC provides to digital analog conversion.

REFERENCES

1 T G HAGEN R TREIBER
Hybrid analog/digital techniques for signal processing applications
AFIPS Conference Proceedings vol 28 pp 379-388 1966

2 R I RUBIN
Hybrid techniques for generation of arbitrary functions
SIMULATION vol 7 no 6 pp 293-308 1967

3 W E CHAPELLE
Hybrid technique for analog function generation
AFIPS Conference Proceedings vol 23 pp 213-227 1963

4 R J JARRETT
An analog R. C. integrator with a digital output
Proceedings of the National Electronics Conference vol XVI pp 611-618 1961

5 A K SUSSKIND
Notes on analog-digital conversion techniques
MIT Technology Press 1957

6 P C DOW
An analysis of certain errors in electron differential analyzers II—capacitor dielectric absorption
IRE Trans. on Elect. Comp. vol EC-7 pp 17-22 March 1968

7 R M HOWE
Design fundamentals of analog computers components
D. Van Nostrand Company Inc Princeton New Jersey 1st ed chap 2 pp 37-41 1961

8 S DRIBAN
Spurious damping in analog computers
Masters Thesis Drexel Institute of Technology pp 49-55 June 1967

9 G A KORN T M KORN
Electronic analog and hybrid computers
McGraw-Hill Book Company New York chap 3 pp 110 1964

PHENO–A new concept of hybrid computing elements

by WOLFGANG GILOI
and HEINZ SOMMER
Technical University
Berlin, Germany

INTRODUCTION

PHENOs are based on the well-known fact that a digital-to-analog converter with variable reference (MDAC) can produce the product of an analog and a digital variable.[1] While for multiplication and division this principle can be used directly, it has to be modified for function generation. In order to obtain a system of computing elements in which any input or output variable can exist in analog or digital form, optionally, DACs and ADCs (analog-to-digital converters) are combined. The straight-line-segment approximation of arbitrary functions is done by splitting the (digital) argument of the function in two parts. The first group of r bits defines the nearest preceding breakpoint, while the second group of (n-r) bits is used for linear interpolation (n being the digital word-length). In a second method of function generation, which is particularly suited for multivariable functions, digital table look-up is combined with analog interpolation. On the base of PHENOs, this procedure provides minimum table look-up execution time and avoids stability problems.

The ADCs are key elements with respect to operation speed. Here, adaptive continuous converters, using the up-down-counter principle, show best performance. Constant band width/accuracy ratio is obtained by automatically adapted register length, i.e. for input increments greater than the least significant bit this bit is dropped, resulting in a doubled step size, etc. The conversion rate is further increased by using a subranging technique.

At the time being, the ADCs operate with 6 MC clock frequency. Therewith, a continuous ADC can track a 100 cps sine-wave with 0.01% accuracy and 1 kc sine-wave with 0.1% accuracy. Static errors of multiplication, division and function break points are less than 0.01%.

PHENO-equipped analog computers will essentially not be more expensive than high-precision analog computers containing very precise time-division or quarter-square multipliers, but they will produce a much better bandwidth/accuracy ratio. Additionally, they require no expensive interface when being linked with digital computers, since the variables are partly existing in digital form, anyhow. On the other side, PHENOs can be used to build inexpensive, small-scale, special-purpose computers for military and process control application. For this, additional elements have been developed, for example, units which the the combination of PHENOs with DDA-elements possible (DDA = digital differential analyzer). As opposite to usual analog computing elements, PHENOs permit miniaturization and operation under difficult environmental conditions.

Principle of PHENO

In a ladder network according to Figure 1 the short-circuit transfer conductance is[2]

$$\frac{i_o}{e_{i1}} = \frac{1}{2R} \sum_{k=1}^{n} \frac{S_k}{2^{n-k}}; \qquad (1)$$

$$S_k = \begin{cases} 0 \text{ (switch connected with ground)} \\ 1 \text{ (switch connected with } e_{i1}) \end{cases}$$

and, therefore, proportional to the digital number $d^* = d_n d_{n-1} \ldots d_2 d_1$ stored in the register. Connecting the output terminals 2-2′ with the input of an operational amplifier with feedback resistor $R_f = R$ results in an amplifier output voltage, which is related to the input voltage by

$$e_o = e_{i1} \cdot d^* \qquad (2)$$

By exchanging the input and output network of the

amplifier we have the inverse relation

$$e_o = \frac{e_{i1}}{d^*} \qquad (3)$$

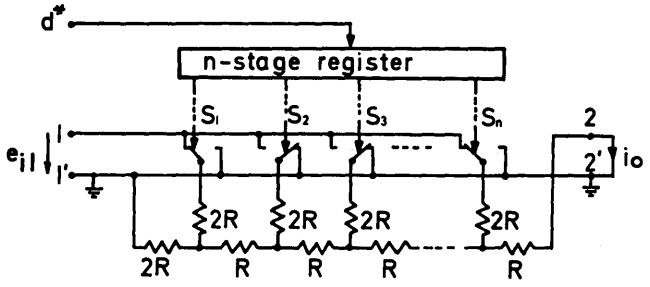


Figure 1—Principle of D/A-conversion

Hence, we obtain in the first case the product of an analog and a digital variable, while in the second case we get the quotient of both variables. In both cases the results are given in analog form. In the following, such a multiplying digital-to-analog converter will be called 'MDAC.'

Most analog-to-digital converters (ADC) are based on the principle that by a certain strategy the register setting d* of a digital-to-analog converter (DAC) has to be found in such a way that the output of the DAC equals the input voltage e_{i2} that has to be converted. When using a MDAC with reference voltage e_{i1}, we can obtain a register setting

$$d^* = K\frac{e_{i2}}{e_{i1}} \qquad (4)$$

i.e., d* gives, in digital form, the result of dividing two analog variables.

It is important to have an element for arbitrary function generation as for multiplication and division. In an analog computer (as well as in digital table look-up programs) the most simple solution of that problem is a straight-line-segment approximation of the desired function. This kind of function generation can be obtained for functions of one (digital) variable by a modified digital-to-analog converter, in which the argument of the function is split into two parts. The first r bits of the digital input of length n define (in binary code) the abscissa at the nearest breakpoint (the function being approximated, for example, by 2^r straight-line segments of equal length), while the remaining (n-r) bits are used for linear interpolation within the actual segment. For that purpose, this second part of the digital input is fed to a MDAC which has the function increment between the two actual breakpoints to reference. Of course, it is just a matter of coding to obtain any other distribution of the breakpoint abscissa. Since this function generating digital-to-analog converter (FGDAC) may operate with variable refrence, ti performs the operation

$$e_o = e_i \cdot f(d^*) \qquad (5)$$

If the output of one FGDAC generating the function $f_1(x^*)$ is used as reference of a second FGDAC generating the function $f_2(y^*)$, we can obtain the expression

$$e_o = e_i\,[f_1(x^*) \cdot f_2(y^*) + a_1 f_1(x^*) + a_2 f_2(y^*)]\ , \qquad (6)$$

which, in many cases, gives a good approximation of functions of two variables f(x,y).

Using a FGDAC as part of an ADC gives the operation

$$d^* = f^{-1}(e_{i2}/e_{i1}) \qquad (7)$$

(f^{-1} being the inverse of the function f(d*) which is generated by the FGDAC).

By combining the operations according to eqs. (2) through (7) (that means combining digital-to-analog and analog-to-digital conversions) multiplication, division, function generation or combinations of these operations can be arbitrarily performed with analog and/or digital inputs and outputs.

The purpose of this new technique is to exceed the accuracy limits of usual nonlinear analog computing elements. Since any operation of the PHENOs is based on conversion techniques, maximum static accuracy is that of high-precision converters. An analog accuracy of 0.01% (single-scale) and a digital resolution of 14 bits plus sign can be achieved. Hence, the PHENOs are well-matched to the static precision of best linear analog computing elements.

The dynamic errors depend on the conversion speed of MDACs and ADCs. The MDACs are not critical, because all switches in Figure 1 are set simultaneously. Even in the more critical case of variable reference switches, their settling time (to an error smaller than 0.01%) can be kept in the range of 1 μs, approximately.

The conversion time of a successive-approximation ADC is a multiple of that, because complete parallelism in the ADC operation is too expensive. Even when using an expensive subranging procedure, a conversion rate of 200,000 per second is the highest that could be achieved till now. This extremely fast successive-approximation ADC would lead to the same phase error as in linear analog elements with 60 kc 3 dB-frequency. The successive-approximation ADC starts each conversion from zero; i.e. it does not matter whether or not the input signals are continuous. Because the PHENOs are assumed to operate continuously, however, we can take advantage from this fact in order to increase conversion speed. An ADC which operates strictly on continuous signals is called a 'continuous converter.'

The most simple principle for continuous converters is that of using up-down-counters. Here, dynamic errors are almost negligible as long as the condition holds

$$\frac{d}{dt}(e_i) < \frac{e_{ref}}{2^n} \cdot f_c \qquad (8)$$

(n being the digital word length, f_c the counter clock frequency and $2^{-n} \cdot e_{ref}$ the step size).

When the input voltage slope is greater than the counter clock-frequency multiplied by the magnitude of the counter increment, the ADC is 'overrated,' resulting in a time-increasing error. f_c depends on the settling time required by the comparator and the analog switches. It is a function of the magnitude of the least significant increment.

It is difficult to find appropriate semiconductors for building analog switches with variable reference and 10^{-4}-accuracy (circuits with alloy chopper-transistors with high v_{BE} reverse voltage or circuits with field effect transistors have a relatively poor bandwidth). On the other hand very fast variable reference switches with 10^{-3}-accuracy are easy to implement. Therefore, it is of advantage to use a subranging technique by dividing the entire input signal range into k equidistant subranges. The actual subrange in which the input is falling is then expanded by a factor k (e.f., k = 16), so that a conversion accuracy of $k \cdot 10^{-4}$ is solely required. When the input crosses a boundary between two adjacent subranges the next one has to be selected. In this paper we will show a special technique by which that can be done within one clock interval. Of course, the comparator and the analog switches can now work much faster since they have to settle only to an error less than $k.10^{-4}$.

An other important measure in order to increase conversion speed is in using an automatically adapted register length; i.e., when the converter is going to be overrated n will be diminished. For example, when an actual input increment exceeds the magnitude of the least significant bit, this bit is dropped, resulting in a doubled step size, etc. By this means a constant bandwidth/accuracy ratio is obtained.

As we shall show in this paper, by using this advanced technique we designed PHENOs which have an accuracy-bandwidth performance suiting the most precise linear analog computing elements and exceeding the nonlinear ones.

Realization of the AD-continuous converter

Subranging

PHENOs operate in a ± 10 V-range. In order to be able to use fast current-switches of 0.1% accuracy, we divide the ± 10 V-range into 16 subranges. Therefore, each subrange covers 1.25 Volts of the input voltage. A schematic block diagram of a subranging converter is shown in Figure 2. In the subranging techniques one has to select, of all subranges, the one that includes the input voltage, e.g., by subtracting all lower subranges from the input signal. By the same operational amplifier the resulting difference is multiplied by the factor 8. Thus, the following subrange-ADC has to operate with the reduced accuracy of 0.1%. Subtraction, subrange-selecting D/A-conversion, and amplifying must be executed with an accuracy of 0.01% (with respect to the input). One subrange contains 1024 steps, which is the range of operations of the fast subrange-ADC. As long as the input remains within a specific subrange, it operates with the 6 mc clock frequency. When running into the adjacent subrange, a new subrange must be provided with an accuracy of 0.01%. This operation takes much more time (some microseconds) than the clock rate of the subranging converter (0.16 μs) gives. Therefore, considerable errors could arise due to the slowly operating 0.01%-accurate elements. The following section outlines measures that eliminate these errors, although slowly operating elements are used.

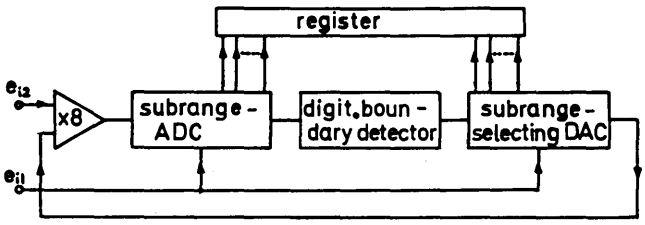


Figure 2—Schematic block diagram of a traditional ADC with subranging

Principles of subranging continuous-converters

When the input signal is crossing the boundary of one subrange, the converter has to switch over to the adjacent one. If the next subrange is prepared by a second subrange-selecting DAC, the required signal for the subranged-ADC is available at any instant. For a special reason (which shall be explained later), the subrange-ADC converts positive and negative input voltages. As long as the voltage is positive the next higher subrange is prepared. If it is negative, the adjacent lower subrange will be prepared.

Preparing of a new subrange is started when the input of the subrange-ADC crosses zero. Therefore, elements with 0.01 %-accuracy have a time interval available for reaching the steady state which is 1024 times the clock-interval of the subrange-ADC. Hence, the subrange-ADC operates over the full scale within subranges which are always in a steady-state, so that no

additional errors can occur. The price for this is in providing a second subrange-selecting DAC. Figure 3 shows the block diagram of the complete continuous converter.

The subrange-selecting DAC 'SSDAC I' is active, when even-numbered subranges are used, and 'SSDAC II' operates on odd-numbered ones. Whether or not a subrange is even-numbered is determined by flipflop FF 1, which switches the subrange-ADC to SSDAC I or SSDAC II. Furthermore, FF 1 marks which system is actually used and which one is in the preparing state. Usually, subranging ADCs, which are known from the literature,[2] have only one subrange-selecting DAC. When the input reaches the upper boundary of a subrange, the subrange-selecting DAC is set to the next higher subrange, and the ADC is reset to the lower boundary of that subrange. If the input voltage decreases or if there is an overflow in the DAC-system, the ADC must be switched back to the subrange just left. This may cause an instable operation, in cases when the input fluctuates around a subrange boundary.

This can be avoided by expanding the range of operation of the subrange-ADC to negative values. Nevertheless, the converter is set back to zero (and not to the lower boundary) when the input crosses the upper boundary. If the input is now decreasing, after just having left a subrange, the converter can use the negative part of the new range. Due to this principle the converter will always remain in the new subrange after a subrange-change and, therefore, stability at the transition-levels is secured.

If for example the subrange-ADC in Figure 4 reaches the value 16, it is reset to zero and 16 is added to the register of the subrange-selecting DAC. If the input decreases immediately after a subrange-change, the subrange-ADC continues to operate in the new subrange. Two special outputs ("Δe" and "i") are provided for a particular application described in section 4. Note that with "△e" we have in analog form the equivalent of the last 10 bits of the digital output, while at terminal "i" we shall get an impulse every time the input e_{i1} is going to cross a subrange boundary.

Automatic step size adaption of the subrange-ADC

Formula 8 of chapter 1 reads:

$$\frac{d}{dt}(e_i) < \frac{e_{ref}}{2^n} \cdot f_c$$

In order to reduce the time-increasing errors of the

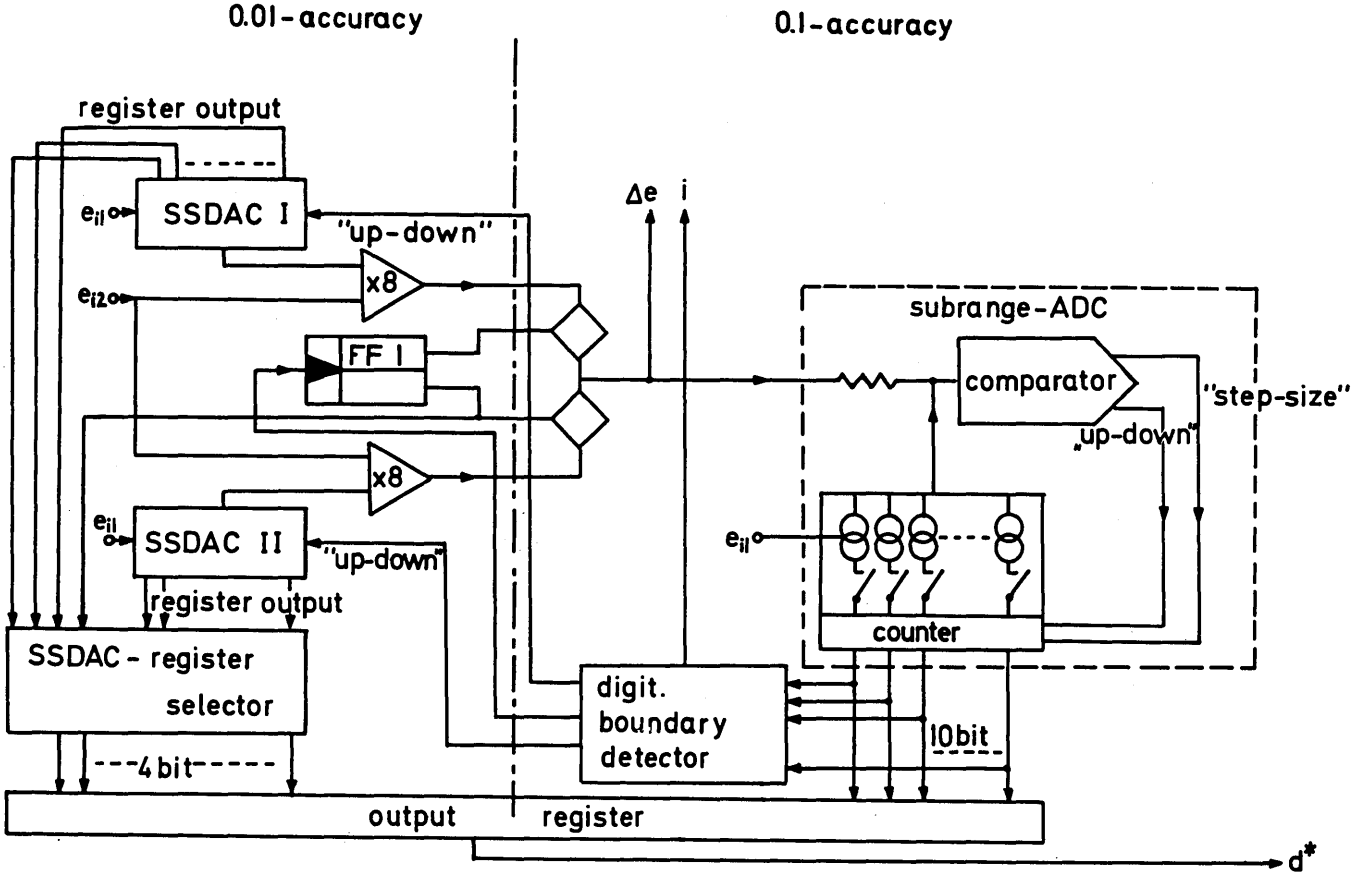


Figure 3—Block diagram of a PHENO-ADC

usual up-down-counter ADC we can either choose a higher clock frequency or a larger step size.

The transient function of most analog switches are almost exponential. Thus, the transient time for 0.01% accuracy is equal to 9 time-constants and the transient time for 0.1%-accuracy is equal to 5 time-constants. By doubling the clock-frequency the error would increase by the factor 100. On the other hand, errors increase only linearily with the step size, so that a constant error bandwidth ratio can be obtained.

The step size $\frac{e_{ref}}{2^n}$ of a counter-ADC depends on the magnitude of the reference voltage and the digital word length (register length) n. The static error of PHENO is less than 1mV. Assuming a 10V-reference voltage, we obtain a relative error of 0.01%. With decreasing reference voltage the step size decreases, too. While errors of the comparators and switches are constant, the maximal slope of the input voltage is reduced.

By an automatically adapted register-length the required step size can be matched to the slope of the input-voltage. The criteria for changing the step size can be obtained by measuring the summing-junction offset of the subrange-ADC. Traditional counter-ADCs detect whether the summing-junction has a positive or negative offset, and thus the counter is set in the 'up' or 'down' mode. Additionally, in our case the amplitude of the summing-junction offset is measured by two additional comparators. According to the used step size, the threshold voltages of these comparators have to be varied. A converter-'overrating' may occur under different circumstances. E.g.: if there is a positive step and

(i) a positive overflow, the stepsize has to be halved
(ii) no overflow, the stepsize has to be retained
(iii) a negative overflow, the stepsize has to be doubled.

For negative steps similar considerations will hold.

Hence, the sign of the last step has to be stored, and, in order to decide how to change the step size, that information has to be combined logically with the outputs of the overrate-detecting comparators.

Our AD-continuous converter has 8 possible step sizes, starting with 1mV. The other step sizes are given by successive doubling. The actual step size is stored in a 3-flipflop counter, which determine the register length and the threshold of the comparators.

Implementation of PHENOS

The elements

In table I existing standard types of PHENOs are listed. In addition to elements for AD- and DA- conversion, multiplication and division as well as function generation, special elements such as incrementing ADCs and accumulating DACs have been developed. They give the possibility of combining PHENOs with digital differential analyzer elements (DDAs). By combining them, new systems can be created, e.g., hybrid integrators, etc.

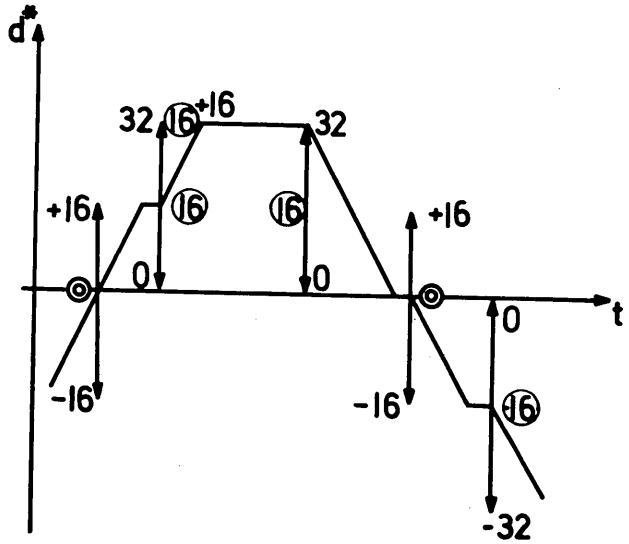


Figure 4—Scheme of subrange-transition
NOTE: The numbers at the arrows indicate the range of operation of the various subranges, the encircled numbers give the setting of the actual SSDAC

It should be emphasized that the possibility of "slaving" several elements also exists. For example, one ADC can be connected with two or more MDACs or FGDACs, giving the possibility of "multichannel" multiplication or function generation.

Analog switches and variable function generator circuitry

The operating speed of MDACs and ADCs depends mainly on the transient time of the analog switches. In our subranging continuous converter two types of analog switches are required: (i) a very fast one with 0.1%-accuracy and (ii) a very accurate switch (0.01%), which is uncritical in terms of speed.

Newly-developed, inexpensive variable reference current switches are meeting these requirements. The transient time of the simple, 0.1%-accurate switch plus comparator (μA 710) is 150 ns. The transient time of 0.01%-accurate switch is about 500 ns.

The principle of function generation has been explained in section 1. The most significant r bits of a digital number with length n define the breakpoints. The remaining (n-r) bits are used for linear interpolation.

The function-ordinates at the breakpoints are adjusted by potentiometers, which are sources with inductive output impedance. The switches must provide that the currents flowing in the potentiometers are not affected by switching (Figure 5).

As Figure 5 shows, the settings of potentiometers $P_1 \ldots P_r$ are not affecting one another. According to the digital number, a decoding matrix causes one pair of FETs, e.g., F_1 and F'_1, to be switched on and the transistors T_1 and T_2 to be switched off. All other switches are used the opposite way. If F_1 is switched on, the voltage $-V_1$ appears at the output of amplifier A_1. At the same time a current (proportional to the difference $(V_2 - V_1)$) flows through the FET'$_2$, resulting in the voltage $-(V_2 - V_1)$ at the output of amplifier A_2. $-(V_2 - V_1)$ acts as reference voltage of a MDAC which linearily interpolates between breakpoints.

The FET-switches 1 through r operate with an accuracy of 0.01%; their settling time for that accuracy is about 2μs. By inserting them in the feedback loop of an operational amplifier, errors due to their temperature-sensitive on-resistance are avoided. The FETs 1′ through r′ switch the difference voltages of two adjacent breakpoints to the summing-junction of the amplifier A_2. Here, the temperature-variable on-resistances of the FETs are part of the resistors which define the slopes of the straight-line segments. This is possible since the maximum function increments between adjacent breakpoints are limited and only small error propagation can occur.

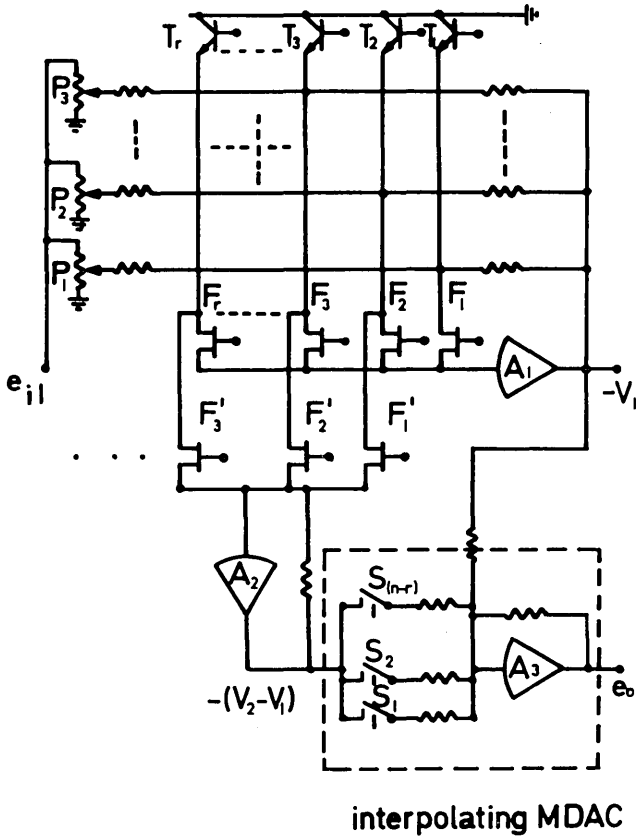


Figure 5—Circuitry of a FGDAC (schematic)

The transistors T_1 through T_r are in on-state when the related FET-switches are switched off. By that measure, discontinuities of the currents flowing in the potentiometers are avoided.

Incrementizer for linking hybrid and DDA sub-systems

DDA-elements are processing increments of digitally represented variables. Generally, these increments only have the values +1, −1, or 0, corresponding to the least-significant bit of the digital word.

When applying that DDA-technique to analog signals, increments smaller than the digital aperture may occur if the rate of change of the analog signal is too small (compared with the clock interval). Thus, the DDA may not give any response at all, though the analog signal can slowly vary over the entire scale. This difficulty can be avoided by accumulating consecutive increments and continuing that until the sum is going to exceed the threshold which corresponds to the least-significant bit. In our counting-ADC this procedure is performed by comparing the sum of all preceding output increments with the actual input. Thus, a counting-ADC can be used for linking analog and DDA-subsystems.

Hybrid integrators

Analog computers perform integration with respect to time. The generalized integration of arbitrary functions can be performed with a hybrid integrator. For this purpose integration is approximated by Euler's formula

$$z(x) = K \int_{x_o}^{x} y\,dx = Z_n(x) \qquad \begin{matrix} x = x(t) \\ y = y(t) \end{matrix} \qquad (9)$$

$$Z_n(x) = K \cdot \sum_{k=o}^{n} y_k \cdot \Delta x_k, \quad \Delta x_k = +1 \text{ or } -1 \qquad (10)$$

Referring to Figure 6, the hybrid integrator consists of two ADCs and one DAC. ADC I produces the increments +1 oder —1 (with respect to the clock period T^*_1). The analog switches S_1 and S_2 generate the product $(y_n \Delta x_n)$ which is added to the sum

$$K \cdot \sum_{k=o}^{n-1} y_k \Delta x_k$$

obtained before. The result is converted by ADC II which works with a higher clock rate. The output of ADC II is stored in DAC I (in digital form) and its analog equivalent is fed back to summer S. The resulting $Z_n(x)$ is available in digital as well as in analog form.

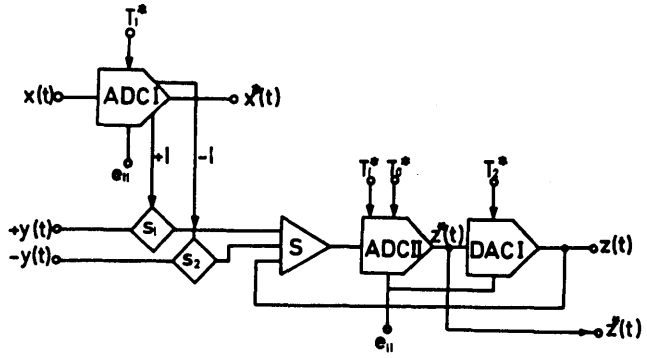


Figure 6—Hybrid integrator, programmed with PHENOs
NOTE: Asterisks denote digital variables

PHENOS in hybrid computer systems

For being linked with a digital computer, PHENO-equipped analog computers only require an inexpensive control interface but no data interface. Since all required analog-to-digital conversions are simultaneously performed by independent continuius converters, no errors due to the time shift of analog multiplexing or to the skewing time of sample-and-hold circuits can occur.[4] Moreover, PHENO-equipped analog computers as part of a combined hybrid computer system offer particular possibilities which do not exist in conventional hybrid systems.

As an example of great practical importance, we consider the problem of multivariable function generation. On the base of PHENOs, the well-known digital table look-up routine can be combined in a simple way with an analog interpolation, resulting in minimum execution time. Because of a direct analog path between input and output signals, the stability problems of pure digital function generation are avoided.

The generation of arbitrary functions is executed by straight-line-segment approximation. For example, in the case of a function of one variable we have the interpolation formula

$$f(x) \simeq f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i}(x - x_i)$$

$$\simeq f_i + m_i(x - x_i) \quad , \qquad (13)$$
$$\text{for } x_i \leq x \leq x_{i+1} \; ; \; f_i = f(x_i) \; .$$

The breakpoint values f_i $(i = 0,1,2 \ldots,B\text{-}1)$, B being the total number of breakpoints (e.g., $B=2^4=16$,) and the average slopes m_i are stored in the memory of the digital computer. The necessary multiplication of (m_i and $x—x_i$) now can be performed by a MDAC. Note that now, contrary to the FGDAC which was described in section 3.2, the slope is given in digital form while the function argument is analog. Hence, for the digital table look-up routine, the argument x first of all has to be converted by an ADC.

Because of the special subrange technique used in the PHENO-ADCs, the last 10 bits of the 14-bit output word are available in analog form. If the remaining 4 most-significant bits are used in order to define the function breakpoint (corresponding with $B = 2^4 = 16$), the last 10 bits or their analog equivalence, respectively, are identical to the increment $(x - x_i)$. Thus, we can use the very simple setup outlined in Figure 7.

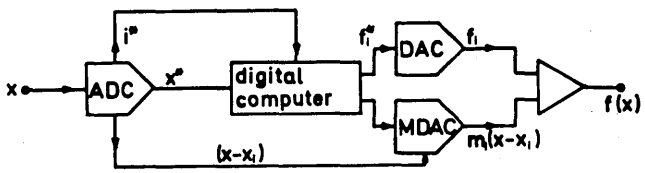


Figure 7—Hybrid generation of functions of one variable on the base of PHENOs

Here, the function argument x is fed into a PHENO-ADC and converted in digital form. Simultaneously,

the ADC gives out the analog signal $(x - x_i)$ which is fed into a MDAC in order to build the product $m^*_i (x - x_i)$.

The digital representation x^* of the function argument is transferred into the digital computer, but only the 4 most significant bits are required for the table look-up routine. By appropriate indexing these four bits can directly be used to give the (relative) addresses of the memory cells in which the corresponding breakpoint abscissa f_i and the average slopes m^*_i can be found.[3] Therefore, the digital program has to execute the following routine:

(i) Load x^* into accumulator;

(ii) Shift x^* to the right so that the most significant 4 bits give the relative address of the memory cell in which f^*_i is stored;

(iii) Load f^*_i into DAC;

(iv) Add B to the address. This modification gives the relative address of the memory cell in which m^*_i is stored.

(v) Load m^*_i into MDAC.

(In the case of the digital computer SDS 930 for which the above routine was programmed, the entire instruction sequence requires about 16 memory cycles $= 28\ \mu s$).

That digital table look-up routine is not necessarily part of the intrinsic hybrid computation. Since the ADC provides an interrupt signal ("i") when the input 'x is going to cross a subrange boundary (which is identical with crossing a function breakpoint), the hybrid computation can be interrupted. While the analog computer remains in the 'hold' mode, the actual values f_i^* and m_i^* can be replaced by f^*_{i+1} and m^*_{i+1}, and then, the hybrid computation may continue. During the real-time operation of the combined system there is an analog path between input and output of the function generation procedure. Hence, the stability problems are avoided which can arise when a digital computer is inserted in analog loops.[4]

We have to emphasize that this procedure is similar to the one previously proposed by A. I. Rubin.[5] But when PHENOs exist, our approach is simpler and less expensive.

The above principle can easily be expanded to the task of generating functions of more than one variable. If, for example, a function of two variables f(x,y) has to be generated, we use the interpolation formula*

$$f(x,y) = f_{i,k} + m^x_i (x - x_i) + m^y_k (y - y_k) \quad , \quad \text{for } x_i \leq x \leq x_{i+1}\ ,\ y_k \leq y \leq y_{k+1} \quad , \quad (14)$$

$$f_{i,k} = f(x_i, y_k)$$

which leads to the implementation given in Figure 8.

Now, for any breakpoint of the two-dimentional grid the values $f^*_{i,k}$, $m^x_i{}^*$, and $m^y_k{}^*$ have to be stored. An appropriate modification of the above outlined special indexing technique leads to a very efficient table look-up routine by which those values can be loaded into the DAC and the two MDACs (requiring now 38 μs on the SDS 930). Note that only one additional pair of ADC and MDAC is necessary for each additional input variable.

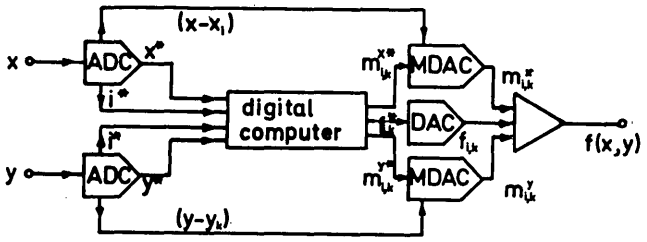


Figure 8—Hybrid generation of functions of two variables

Hybrid special purpose computers

PHENO offers a particular flexibility in building precise, inexpensive, miniaturized, hybrid special purpose computers. If only 'static' operations have to be performed (such as summing and subtraction, multiplication and division, function generation), a combination of PHENOs and analog summing amplifiers will be sufficient. The accuracy is strictly dependent on resistors, electronic switches, and operational amplifiers. Since all these components can be realized with small temperature drift, operation under extreme environmental conditions is possible. A high degree of integration and, therewith, miniaturization can be obtained.

In the case of 'dynamic' problems, i.e., problems which include integration, either the hybrid integrator of section 3.4 may be used or a combination of PHENOs with DDA-elements. The first approach is advisable if only a few integrations are required. For the second approach we shall give a typical example.

In case of a special purpose computer for solving a proportional navigation problem, the solution of the following equations is required

$$d\gamma_M = K \cdot d\sigma \quad (15)$$

$$K = K_o f(e) \quad (16)$$

$$\gamma_M = \Sigma d\gamma \quad (17)$$

$$du_1 = -u_2 d\gamma_M \quad (18)$$

$$du_2 = u_1 d\gamma_M \quad (19)$$

$$dV_y = V_M du_1 + u_1 dv_M \quad (20)$$

$$dV_x = V_M du_2 + u_2 dV_M \quad (21)$$

*Rubin has used a more sophisticated interpolation formula which gives a slightly better approximation but requires a more expensive hardware.

We assume that the input variables $d\sigma$ and dV_M are given as digital increments, the input e and the output γ_M are digital parallel words, and the output variables V_x and V_y are required as analog signals. Therefore, we may rewrite eqs. (16)-(21) as follows, denoting digital parallel information by asterisks.

	Components
$d\gamma_M = K^* \cdot d\sigma$	DDA-integrator
$K = K_o \cdot f(e^*)$	FGDAC
$K^* = K \cdot 1$	ADC
$\gamma^*_M = \Sigma d\gamma_M$	counter
$du_1 = -u_2 d\gamma_M$	DDA-integrator
$du_2 = u_1 d\gamma_M$	DDA-integrator
$dV_y = V_M du_1 + u_1 dV_M$	DDA-integrator
$dV_x = V_M du_2 + u_2 dV_M$	DDA-integrator
$V_y = \Sigma dV_y$	accumulating-DAC
$V_x = \Sigma dV_x$	accumulating-DAC

Figure 9 shows the resulting computer block diagram on the base of PHENOs according to Table 1 and DDA-components.

REFERENCES

1 T G HAGAN
AMBILOG computers: hybrid machines for measurement-system calculation tasks
Proc. 17th Annual ISA conf New York Oct 1962

2 DIGITAL EQUIPMENT CORPORATION
Analog/digital conversion handbook
Maynard Massachusetts 1964

3 G SCHWEIZER H SEELMANN
The application of hybrid simulation for VTOL-aircraft and certain reentry-problems
Proc. 4th Internat. Analogue Computation Meetings Brighton 1964 pp. 18-29

4 W GILOI
Error-corrected operation of hybrid computer systems
Paper presented at the 5th Internat. Analogue Computation Meeting Lausanne 1967

5 A I RUBIN
Hybrid techniques for generation of arbitrary functions
SIMULATION vol 7 no 6 Dec 1966 pp. 293-308

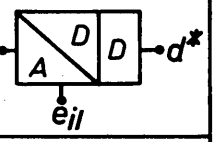
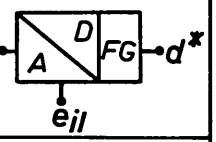
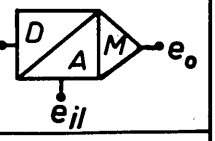
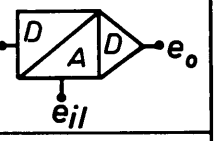
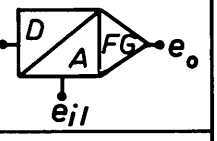
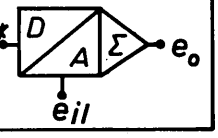
element	formula	symbol
ADC	$d^* = \frac{e_{i2}}{e_{il}}$	
FGADC	$d^* = f(\frac{e_{i2}}{e_{il}})$	
MDAC	$e_o = e_{il} \cdot d^*$	
DAC	$e = \frac{e_{il}}{d^*}$	
FGDAC	$e_o = e_{il} \cdot f(d^*)$	
Accumulat.-DAC	$e_o = e_{il} \cdot \sum_{K=1}^{n} \Delta d_K^*$	

Table I

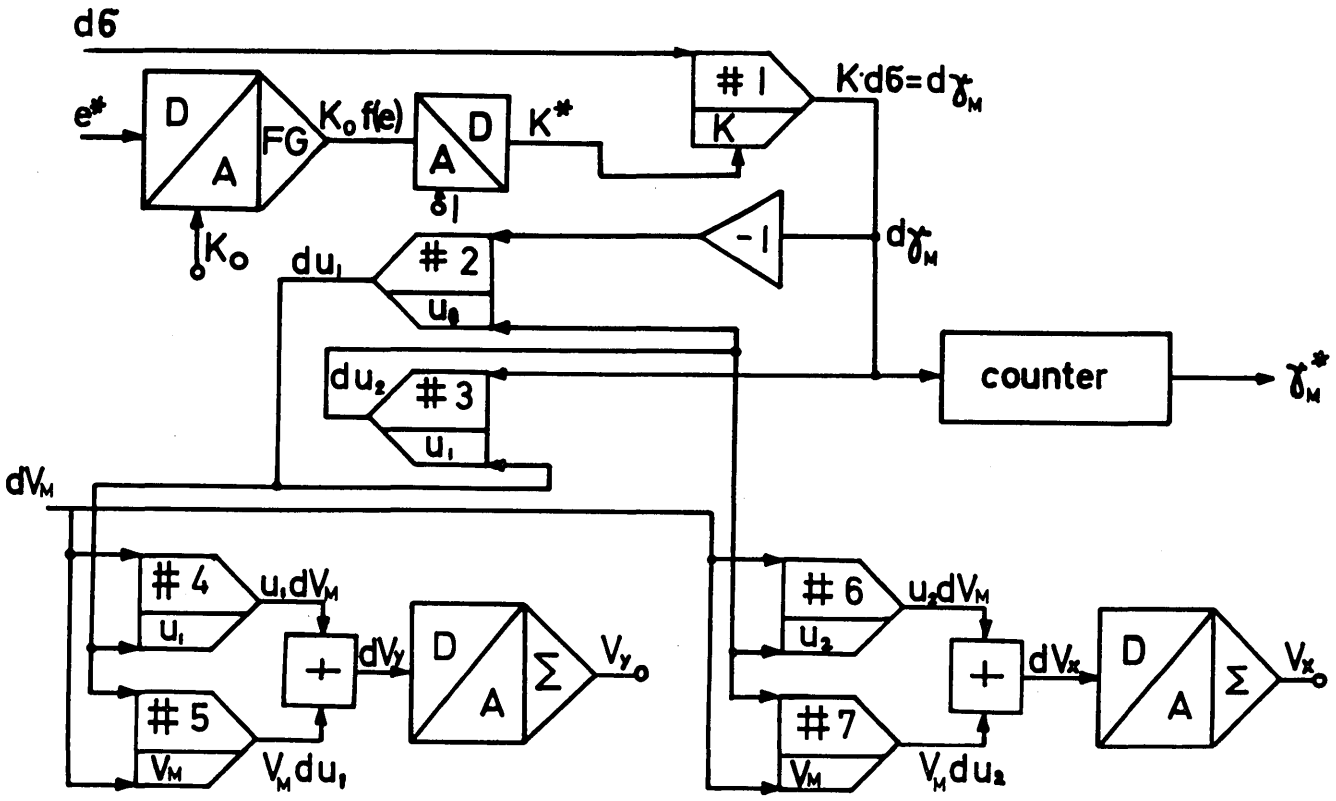


Figure 9—Block diagram of a special purpose computer, solving eqs. (13)-(19). The computer consists of PHENOs and DDA-elements

Textile graphics applied to textile printing

by JANICE R. LOURIE and JOHN J. LORENZO
International Business Machines Corporation
New York, New York

INTRODUCTION

Description of textile graphics

Textile Graphics is a computer-aided technique[1] for developing a textile design and textile patterning mechanism information from an artist's drawing. The computer is operated by a textile designer-technician who understands the particular textile machinery for which he is adapting the original drawing. The designer-technician inputs the original drawing by a combination of graphical input devices; tracing on an on-line digitizing tablet, drawing free-hand with light-pen on the screen of the IBM 2250 and manipulating the design with function keys.

After the original design is in core it is developed into the information to control the patterning mechanism of a specific kind of textile machinery. For example, a design to be woven must represent each *interlacing* of warp and weft; a design to be knitted must represent each *stitch* of the knitted mesh; and a design to be printed must represent the areas of each *color* as separate images.

The development of this information is done according to both structural and aesthetic rules. The designer-technician interacts with the computer using the function keys, light-pen, 2250 and photographic plotter.

When the design control information is developed, it may be outputted under function key control. The form of the output is commensurate with a specific patterning mechanism. For example, for a Jacquard loom which is controlled by punched cards, Textile Graphics output is the pattern of the holes in these cards; for a Raschel machine which is controlled by a chain of cam-links, Textile Graphics output is a map of the heights of the successive links; and for textile printing machines which are controlled by etched copper rollers or silk screens, Textile Graphics output is a set of color-separated films.

Previous applications of textile graphics

The first application of Textile Graphics was to Jacquard weaving.[1] The Jacquard loom is controlled by punched cards and the designs which it produces may be described in computer terms as rectangular (mxn) binary matrices with a "1" in any position. The size of m and n may vary from "one" to several thousand.

This direct analogy between the representation of computer data and textile design information was noted in a description of Charles Babbage's Analytical Engine. It said that his "computer" weaves algebraic equations like a Jacquard loom weaves flowers. It is this analogy which motivated the original work in Textile Graphics.[2]

Textile Graphics was then applied to developing design control information for lace which is manufactured on a Raschel machine. Although the Raschel machine is a warp knitting machine and not a weaving machine like the Jacquard loom, its design control information is currently developed on a grid paper analogous to the point paper described in Ref. 1. This practice permitted the natural extension of Textile Graphics to lace and other knitted fabric design.

A new application of textile graphics

This paper describes the application of Textile Graphics to textile printing. The application to textile printing differs from applications to other forms of textile design because the printed textile design is applied after the textile is fabricated.

Since the patterning mechanism is independent of the fabric forming mechanism, there is no need to represent the design on a grid paper, (which for other textiles is translated into row by row, design forming information). However, we will show that for a large class of designs, the color-separated images which printed textile designs require can be more efficiently produced using Textile Graphics.

First, the major current methods of producing printed textile designs are presented. Then, the types of designs are analyzed and reclassified to give more insight into new methods of obtaining color separations.

We combine different parts of current methods with Textile Graphics to expedite the production of the design control information (color separations) for approximately half of printed textile designs. In addition, a new category of printed textiles, whose color separations are easily generated using Textile Graphics, is exhibited.

Current method of preparing designs for textile printing

Description of textile printing

There are two major methods of printing designs on textiles. One of these, a roller method, uses etched copper rollers to apply the paint directly to the fabric. The area to be printed is etched in intaglio and the remaining portions are left intact. A separate roller is required for each color.

The other method, a screen process, uses fine silk screens through which the paint is "squeegeed" onto the fabric. Here, the area to be printed is left intact; the remaining portions are masked. A separate screen is required for each color.

Since the roller printing method accounts for the bulk of textile printing production, we will illustrate the application of Textile Graphics to this method. There are analogous procedures for the screen method.

Designs to be etched into copper rollers are transferred to the rollers in one of two ways. Either they are traced, with a stylus, into a coating on the roller, or, a film bearing the design is wrapped around the roller and then photographically exposed. The first of these processes is called the "pantographic process"; the second is called the "photographic process." Each of them will now be described in greater detail.

An outline of the pantographic process

The starting point for producing a copper roller by the pantographic process is an artist's sketch. The first step in this process begins by putting the artist's sketch in a magic-lantern-like device and projecting it, enlarged, on a coated zinc plate. The projection is then traced with a sharp stylus which incises the design into the plate. The people who perform this operation are called sketchers and are the most skilled and use the most ingenuity in the pantographic operation.

During this process "improvement" may be made to the original design. For example, lines intended to be made symmetric or with uniform thickness, which were not made precisely so by the artist, will at this stage be made so.

In addition, there may be a special treatment along the boundary of two adjacent colors. Frequently an extra line is placed at a specific distance from the boundary. This line, called a double line, prevents adjacent colors from running together. This double line does not actually appear on the printed fabric but is put on the zinc plate and later traced by stylus onto the copper rollers.

After the zinc plate is prepared, it is given to another person called a tracer. The tracer puts the plate on the flat table-like surface of a machine called a pantograph. The pantograph machine has a stylus with which the tracer will follow lines incised into the zinc plate. As the tracer moves this stylus along the groove in the zinc plate, a row of styli are simultaneously moved. These other styli are tracing the design onto a coated copper roller with the appropriate degree of reduction and number of repetitions across the width of the roller. The areas of the zinc plate have been color coded, and the outlines of each color are traced onto a separate roller.

In addition to the original outline which the artist drew, and the double boundaries mentioned above, other information is imparted and traced onto the copper roller. For example, a sizeable flat area will be incised with parallel, diagonal lines called a *ground.* When to use ground lines is part of the knowledge of the engraver in the shop who knows that the paint must be kept at a certain level in the area to prevent blotching and running during the printing process. Therefore, he will specify to the tracer the density of the lines within each area. A device called an indexing drum is put on the pantograph and aids the tracer in placing these ground lines into the appropriate areas.

Another addition to the original artist's information is called *slashing:* the design is placed on the roller so that lines which are perpendicular to the edge of the roller will be slightly angled. The reason for this is so that the metal doctor blade, which squeegees the paint, will not tend to break down the edges (boundary outline) of the roller. Slashing is anticipated in the initial set-up of the styli which trace on the roller. Each successive stylus is progressively displaced along the circumference of the roller.

After tracing, a roller is sent to a touch up bench where it is reviewed and imperfections are corrected before the etching process. The roller is then dipped in acid to etch lines into it. Lines which are to be

etched deeper go through more than one etching stage. Between these stages, certain areas are painted out. After the rollers are etched, they are chrome-plated.

The photographic etching method

Implicit in the discussion of the pantographic method was the assumption that the original artist's design could indeed be conveyed by tracing. This is certainly not true of all original artist's designs. Many of them, which have been created by brush effects, and so forth, cannot be recreated by a stylus alone. Therefore, other techniques are necessary to transfer the original art work to the copper rollers. These other techniques are divided into three major categories: 1. those which are purely photographic; 2. those which are purely manual; 3. those which are a combination of photographic and manual. The object of each of these processes is to produce color separation films—that is, a separate film for each color used in the printing process. This is not a separation into red, yellow, blue, and perhaps black. When the textiles are printed, the color which actually appears on the fabric is the color of the paint used in printing. It is not a mixture at printing time of the primary colors.

After these color-separated films have been made, the process for etching the design onto the drum is the same for each of the three techniques named. Therefore, we will discuss this common etching process after the individual processes for preparing color-separated films.

Photographic etching Type I—purely photographic

In cases where it is desirable to reproduce the original artist's drawing exactly in the printed textile, and furthermore, where the number of colors in the design as well as their nature is such that each color can be separated by a series of photographs with appropriate color filters, then the purely photographic Type I process is employed. This is, however, applicable in a minority of the cases. In the purely photographic method, the original design is photographed once for each color in the design. A filter is used to remove all of one color from the design. The resulting film represents the filtered color separated from the other colors. This part of the process produces one film for each color.

Photographic etching Type II—purely manual

As in the case with the reproductions of most art work in textiles, and also true in some wall paper and ceramic designs as well, the original art work is not of an acceptable quality for direct reproduction. The reasons for this are many, but, for example, a simple line which may be of varying degrees of thickness in the original art work must be more carefully drawn to have only a single degree of thickness for the reproduction process. Even the artist would acknowledge this and it is not intent which produces the original uneveness but rather the carrying out, on impulse, of the idea. This kind of inaccuracy seems to be a necessary concomitant of the artistic expression. The restraint and precision necessary for the reproducible design is not and cannot be present in the original creation. There have been many attempts to constrain the artist as he works, constraining him to particular colors or to work with a precision with which he does not now work. These attempts have been unsuccessful, and we believe it is because of something inherent in the original creative process. Therefore, the majority of original designs must be manually recreated.

In this manual method, the art work is placed on a table with light penetrating it from below. A clear acetate film is placed on top of the artist's drawing. Another person, also called an artist in the mill art shop, prepares an acetate mask for each color of the original design. He may prepare an acetate copy of an original color using brushes. He is of course doing his best to reproduce exactly what he sees shining through the light table. However, since this is a copy, it is to some extent an interpretation, no matter how faithful he attempts to be.

Type III—A combination of photographic and manual methods

Some of the manually painted acetate masks may be negatives of certain colors, others may be direct copies, hence positives. By photographically combining negative and positive films in different ways, it is possible to create masks of other areas.

In the simple example shown in Figure 1,

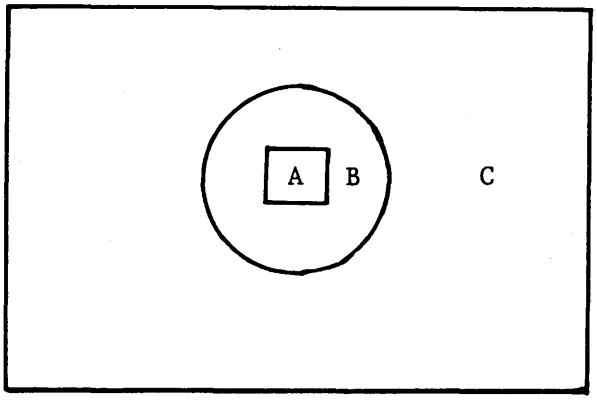


Figure 1—Color "B" is the negative of colors "A" and "C"

if one creates films for A and C, then a film for B can be created by photographing A superimposed on C.

This third category may use some of the original art work to photograph through acetate masks. For example, if there is brushwork in the original which is directly reproducible then it may be photographed and combined with acetate masks of other areas.

Choosing method I, II or III

There are cases when it would appear that a purely photographic method is applicable. However, in many cases, there may be an additional consideration; namely, the design will not evenly fit into the basic repeat size of the fabric on the printing machine. Therefore, it must be reduced or expanded by a certain percentage. Although the camera can be adjusted to perform this reduction or expansion, nevertheless, this results in producing a striated effect on the negative rendering it inapplicable for direct photographic reproduction. In cases such as this, a design which seems to be of the purely photographic type may actually be of the purely manual type.

Common processes to types I, II, III – photographic etching

After the original color-separated films have been made by one of the three methods above, each film is used to produce what is called a *long film.* The long film bears the original design with as many repeats of it as are necessary to fully cover the roller and is the same size as the copper roller, that is, the same width and as long as the circumference of the roller. This long film is wrapped around a copper roller which is then photographically exposed and etched.

In order to make the long film, the individual color-separated films are put, one at a time, into a machine called a "step and repeat" machine. The data which accompany the original color separation film to this machine consist of the number of times the design will have to be repeated on the drum, and the placement of each repeat. There are step and repeat machines which accommodate these data in some automatic method such as a punch tape control. Other models set the data manually. However, the purpose of the machine is to subject the color-separated film to a number of exposures in different positions. They will be developed on the single "long film." After the long film is developed, it is sent to a room for inspection and correction before it is wrapped around the copper roller and exposed. The roller is then etched in acid.

Applying textile graphics

Categories of printed textile designs

From the previous descriptions of the two basic methods of preparing copper rollers, it seems reasonable to divide textile print designs into two categories: traceable designs and non-traceable designs. Traceable designs are those whose boundaries can be created or recreated (traced) by a stylus. The areas inside the boundaries are characterized by flat painted effects which could be made by a brush although they might as well be made by a device such as a felt-tip pen. Non-traceable designs are those which do not possess these properties. They are characterized by brush and stipple effects.

It should be obvious that traceable designs could be produced on a copper roller by either the pantographic process or the photographic process as these two methods now exist. Indeed what we call traceable designs are sometimes now produced by either process, so that calling such a design by the name "pantographic type" as the industry now does is an artificial distinction.

Using textile graphics for traceable designs

Textile Graphics may be used to develop long films for traceable designs which constitute about half of printed textile designs. The input to the computer is an artist's sketch and the output is a series of color-separated long films.

There are three phases of this computer-aided application: input, development and output. During the input phase, part or all of the original artist's drawing is traced on a digitizing tablet; any untraced portions are built up with "symmetry" and other function keys. The regularizing of the design (described above), which is currently done by the sketcher, is done in this phase.

In the development phase, the designer-technician interacts with the computer to develop the additional information needed for printing the design. This information, (described above) which consists of *color separations, double boundary lines, ground lines,* and *slashing* is currently divided between the sketching and tracing phases of the pantographic operation, or between the painting and photographing phases of the photographic operation. The purpose of this development phase is to develop one complete repeat of the design for each color.

During the output phase, the designer specifies the "step and repeat" information to the computer and a series of long films is outputted on the photographic plotter.

Input—phase I

Because a great deal of regularizing of the original artist's sketch takes place during the tracing phase or, what is called in the printing mill, the sketching phase, it is necessary to retain this tracing in any computer assisted process.

The artist's sketch is entered into the computer by tracing it on a digitizing device. The tracing styli of commercially available digitizers are electrostatically or capacitively coupled to a flat surface upon which the art work is positioned. As the tracer moves the pencil-like stylus along the boundaries of the design, this outline coordinate data are transmitted into the computer. The data are collected in the computer's main storage and simultaneously displayed on the IBM 2250 display screen. The 2250 also has a keyboard of "function keys." When the operator depresses one of the keys, the computer program determines which key was depressed and branches to the appropriate section of the program to perform the requested "function." Using these function keys and the light-pen, the designer-operator can translate, enlarge, reduce, erase, or repeat part or all of the displayed design. If the design has symmetry, the operator may elect to trace only a portion of it and request the computer to generate and display the remainder reflected about any chosen lines of symmetry.

The advantages of this input technique over both the tracing that is performed in the pantographic method and the painting that is carried out in the photographic method are significant. Less care and precision is required of the operator (sketcher) because the program "smooths" the free hand tracing. Even assuming no data smoothing, the tracing is physically easier. It requires no more pressure on the stylus than that normally used for pen or pencil on paper, while in the pantographic process, the sketcher must *inscribe* the design into a zinc plate, and in the photographic process he must *paint* the outline on film with brush and ink.

The on-line graphic devices speed up the tracing phase of the process for a number of other reasons:

(1) The pantographic sketcher is working on a design at a fixed enlargement. The simpler areas of a design are unnecessarily traced at this enlargement. Conversely, the photographic sketcher is working with a design at actual size, so that the complex areas of the design must be painted at actual size. The computer-aided approach permits the sketcher to trace various sections of the design at different degrees of enlargement. Under function key and light pen control, the completed sections are then overlapped and given a common enlargement factor.

(2) When there is symmetry present in the artist's sketch, the sketcher may need to trace only a portion of the design. The computer program may calculate and display the remaining symmetrical portions.

(3) Any regular shapes, e.g., squares, rectangles, and circles, present in the design would not have to be traced at all. The desired shape is retrieved from an expandable library of frequently used shapes stored in the computer, and displayed on the CRT. It is given the required degree of enlargement and then rotated and translated into position within the design.

(4) Sketching errors are easily corrected with the light pen.

(5) It may be clearly the artist's intent that lines of varying thickness in his sketch should actually be of constant thickness. The sketcher, using brush and ink, would find this operation tedious, at best. Using light-pen and function keys, the sketcher maintains uniform line thickness of any desired amount.

(6) Both the tracer operating the pantograph machine and the sketcher painting acetate film will circumscribe the entire outline twice. In the pantographic process, the boundary line separating two colors will be traced on each of their respective rollers, while in the photographic process, the boundary line separating two colors, will delimit each area painted on their respective films. Tracing on a tablet will eliminate all of the former tracing operation and half of the latter.

(7) In pantographic tracing, if the length of the basic design is a submultiple of the roller circumference (and it frequently is), additional complete tracings are required. A digitizing tablet eliminates this requirement.

Development—phase II

When the outline of the design has been completely transmitted to the computer, a function key will initiate an algorithm which assigns a unique label to each disjoint area of the design (Figure 2), and displays the labeled areas on the 2250. This algorithm is described in detail in Ref. 1. It performs a raster scan of the design and assigns a zone number to each row segment between "strikes." Every zone is checked against the preceding row to determine if there is any vertical overlapping. If there is, the current zone is given the same label as the zone with which it has common columns. By successive applications of this process, and reassigning labels when it becomes apparent that two zones are equal to each other, it is possible to separate all disjoint areas.

In Figure 3, when the scan reaches row (n + m) it detects that the entire row is one zone. This row-zone

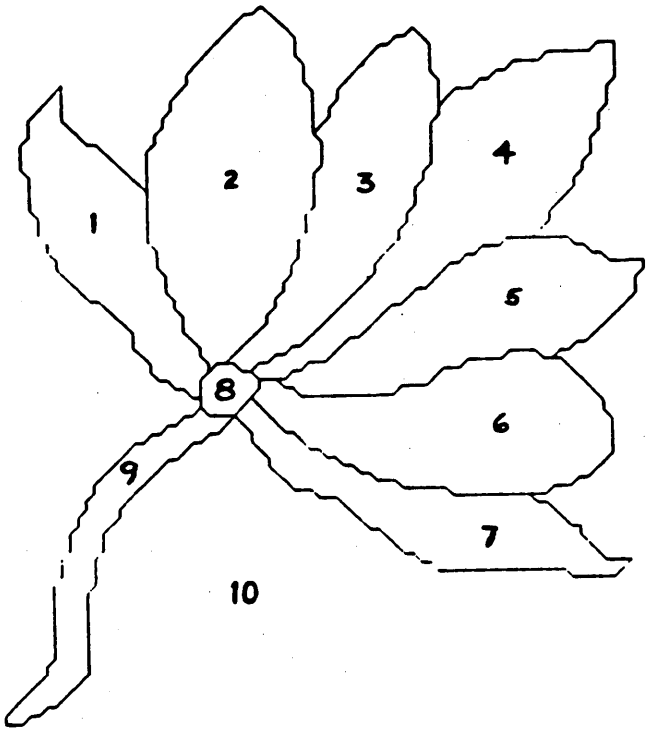


Figure 2 – Design with each disjoint area assigned a unique label

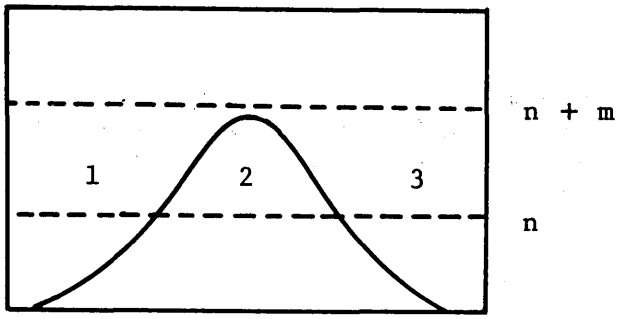


Figure 3 – Example of zone merging

is the same as zone 1 because no curve has intervened. On the other hand, the zone of this row is the same as zone 3 for the same reason. Therefore, the algorithm concludes that zone 1 and zone 3 are in the same connected area. A table of zone equivalences is kept, and after the entire design is scanned, each row-zone is assigned its unique area label.

Figure 4 is the inverse of Figure 3.

It should be noted that this algorithm does not make demands on the user to label his own areas as he creates them, a common procedure in some graphics applications. The purpose of our approach is to permit the designer to operate as naturally as possible.

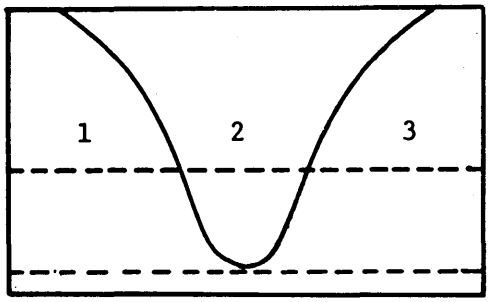


Figure 4 – Example of zone diverging

Colors are *separated* in the following way: the designer uses a function key to enter the "EQUATE" mode. Then he detects, with the light-pen, the labels of all areas which he wants to be the same color. These labels all become the same as the first label detected. A separate file is created containing the set of boundaries for this color. This process is repeated for each color.

Some areas of different color, which have a common printed boundary may actually have such *boundaries* etched into the respective color rollers at a slight displacement. This displacement prevents "bleeding" of the paint during printing. Such a displacement can be specified on the total design by specifying a *double line* at such places. The designer specifies, with function keys and light-pen, the areas and the amount of displacement between their common boundary. Then a double line is created along this boundary which will give rise to the proper displacement on the rollers.

The *ground lines*, which are parallel, diagonal lines within the boundaries of sizeable areas, are used to prevent the paint from running out of the intaglio area on the roller. The areas which need such lines are determined by an engraver and specified by him on the original drawing. The designer-technician transmits this information: name of area, angle of lines and distance between lines – with function keys and light-pen. He may display the design on the 2250 either with or without these lines depending on whether he is looking to make a technical or an aesthetic judgment.

The engraver also specifies the *slashing* angle of the design. The designer, who transmits this, may also view the design with or without this "angling."

Output – phase III

The output phase of the process is the generation of "long films." These are efficiently produced by an on-line high speed photographic plotter. The film is wrapped around a cylinder and the cylinder is rotated

under a CRT. The scanning electron beam of the CRT is digitally modulated, resulting in a plotting time that is independent of the density of the information plotted.

The computer program was initially supplied with the exact width and circumference of the copper rollers to be used. The basic design has been exactly scaled and adjusted to provide the specified number of repeats, both vertically and horizontally, that will completely cover the roller.

A separate film is now plotted for each roller in the design. For example, the design shown in Figure 5 will

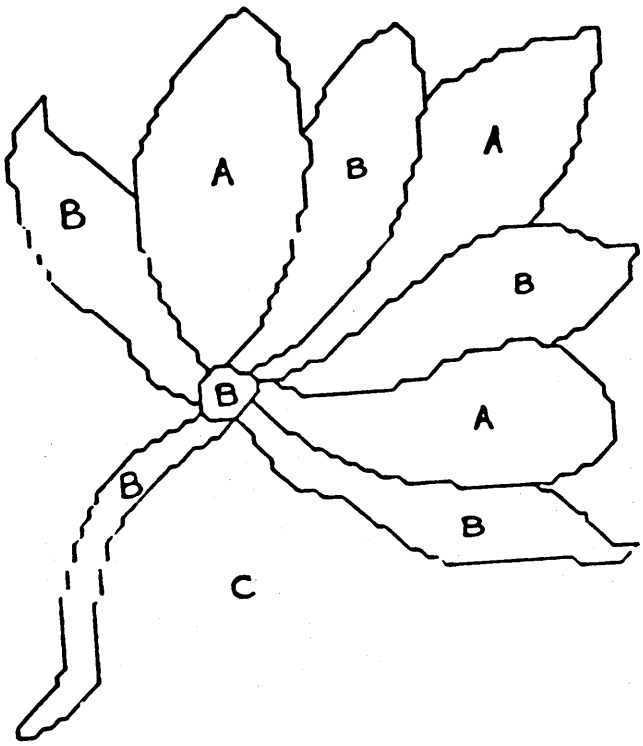


Figure 5 – Three color design

result in the color separated films shown in Figure 6. The double line, ground lines, and slashing (which are not shown in the figure) would, of course, be present. Using the on-line plotter to automatically produce the "stepped and repeated" long films has eliminated the set-up time and possible errors that result when using either the engraving machine in the pantographic process or the "step and repeat" machine in the photographic process.

Looking ahead to a "second generation" textile graphics system, it appears feasible to replace the film and cylinder of an on-line plotter with the copper rollers and to directly photo-expose the design.

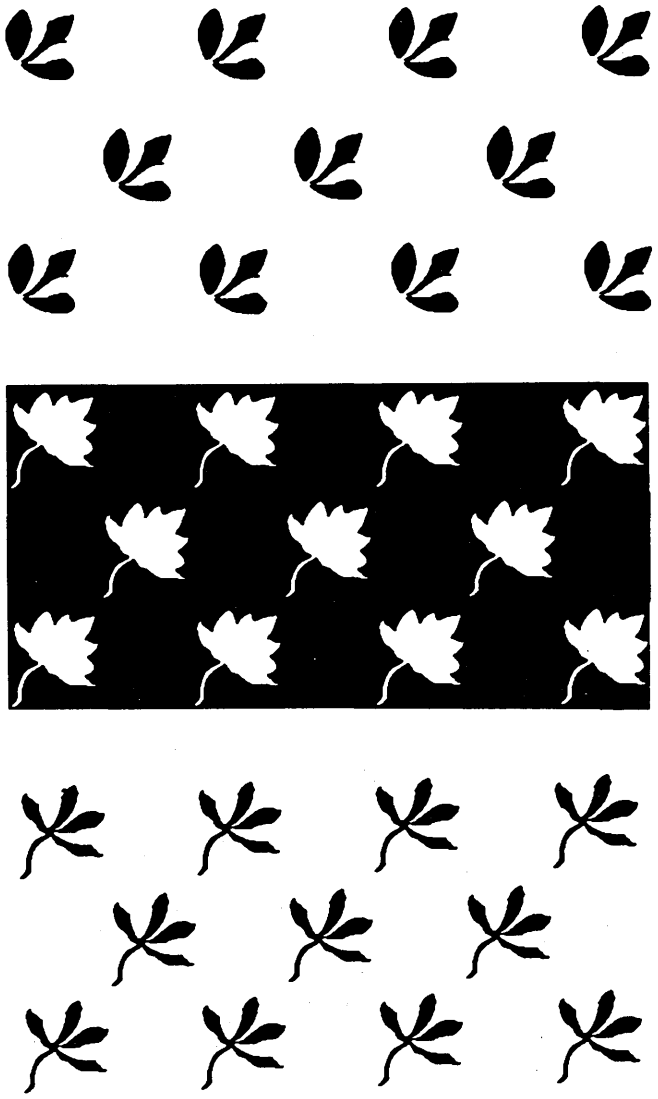


Figure 6a,b,c – The three color separated films of design shown in Figure 5

A new designing tool

With Textile Graphics it is possible for the designer to do something which he has not been able to do before.

Both the photographic etching process and the pantographic etching process have a library of "effects." For the pantographic process this is a set of small round mills which are pressed onto the roller over its entire surface; and for the photographic process this is a library of negatives with repetitive patterns. Both methods use these libraries to generate an entire copper roller with this effect repeated over it.

We store such a "library" within the computer. While sitting at the 2250 console, the designer can specify any of these "effects" to occur within any disjoint area which he points to on the screen. This

greatly enhances his ability to generate patterns since he can use these effects like a Jacquard designer uses "weaves."[1]

An illustration of this technique is shown in the following figures.[7]

The outline of the design is the same as Figure 5. The design is to be printed in four colors. Areas 1, 4, 7 are color #A, areas 2, 6, 8, 9 are color #B, area 3, 5, color #C. The background, area 10, is color D. Note that in areas 1, 4, 7 there is an "effect" which is also color D. This is represented in figure 7.

SUMMARY

We have demonstrated a computer-aided method for generating color-separated "long films" for traceable textile designs, starting with an artist's drawing. These films are the direct input to the roller etching process.

This method merges parts of two processes which are almost distinct at the present time. Namely, the tracing of the pantographic process is maintained, but the etching is continued by the photographic process.

This method is a new application and extension of the Textile Graphics techniques described in Ref. 1. Previously, Textile Graphics had been applied only to Jacquard weaving and to warp-knitting.

We have shown that Textile Graphics can significantly shorten the lead time between the creation of the original art work and the etching of copper rollers. The next logical step is to tie the technique into a fully integrated system monitoring the other operations in the mill. For example, the colors specified for the design, and the anticipated yardage to be produced using this design, would be inputs to inventory control and production scheduling programs. Raw material requirements, finished goods inventory, costing, and printing machine down-time analyses are other areas of importance that can be made available to the textile manufacturer on demand.

REFERENCES

1 J R LOURIE J J LORENZO A BOMBERAULT
On-line textile designing
Proceedings of ACM 1966 National Conference p 537

2 J R LOURIE
The textile designer of the future
Handweaver and Craftsman p 8 Winter 1966

3 A REISFELD
Warp knit engineering
National Knitted Outerwear Association 1966

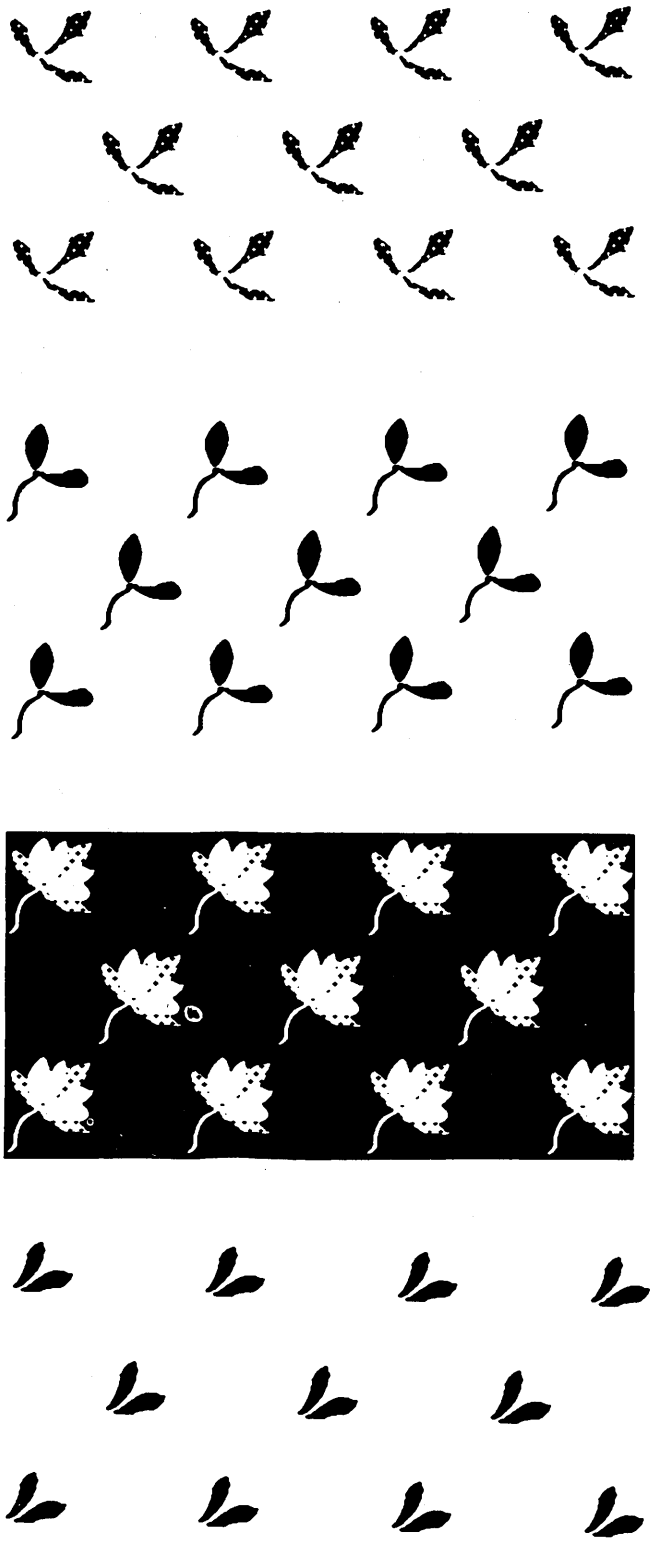


Figure 7a,b,c,d – Color separated films for four color design showing "effects"

Holographic display of digital images

by L. B. LESEM and P. M. HIRSCH
IBM Scientific Center
Houston, Texas
and
J. A. JORDAN, JR.
Rice University
Houston, Texas

INTRODUCTION

An optical hologram is a two-dimensional photographic plate which preserves information about the wavefront of coherent light which is diffracted from an object and is incident upon the plate. A properly illuminated hologram yields a three-dimensional wavefront identical to that from the original object, and thus the observed image is an exact reconstruction of the object. The observed image has all of the usual optical properties associated with real three-dimensional objects; e.g., parallax and perspective.

The computer-simulation of the holographic process potentially provides a powerful tool to enhance the effectiveness of optical holography. The construction of holograms in a computer provides a potential display device. For example, mathematical surfaces can be displayed, thus permitting design to be visualized in three dimensions. In addition, computer generation of holograms can provide a new optical element (i.e., lenses, apertures, filters, mirrors, photographic plates, etc.) without introducing the aberrations usually associated with such elements. Thus "perfect" systems can be examined, and experimental parameters can be easily varied. Also, a computer can perform operations, such as division, which are difficult or impossible, using real optical elements. Furthermore, computer holography can be used to filter the degrading effects of emulsion thickness and non-uniform illumination from optical holograms.

The computer generation of holograms is part of a larger problem. Interpretation of holograms; i.e., the construction of digital images from holograms, constitutes the "inverse" problem. Together, these two processes hold considerable promise. One can construct computer techniques which would take an acoustic hologram (the wavefront from a scattered sound wave) and transform it into a optical hologram, thereby allowing us to construct the three-dimensional image of the scatterer of the sound waves. The same procedures would translate a radar hologram into an optical one. These processes may also make possible magnification of microscopic images in the computer; the optical hologram of a microscopic object could be translated so as to yield a greatly magnified image. Such an image may well be free of the aberrations which so plague three-dimensional microscopy.

The fulfillment of these ideas depends upon the solution of several problems which have inhibited the development of computer-simulated holograms. Efficient computational techniques must be utilized in order that large-scale holograms may be generated in reasonable amounts of computer time. In this paper, we discuss first the mathematical representation of holography, then the computational techniques which we have used, and finally we exhibit some of the results of our efforts.

Mathematical representation of the holographic process

Consider a monochromatic wave from a point source P_o, which is incident upon an aperature A in a plane opaque screen. Suppose the screen is a distance r from P_o and let the point P lie a distance s from the screen. Using Kirchoff's boundary conditions (Born and Wolf[1]) one obtains the so called Kirchoff diffraction formula

$$U(P) = B \int_A \int \frac{e^{ik(r+s)}}{r+s} [\cos(n,r) - \cos(n,s)] \, dA \qquad (1)$$

where B is constant which depends on the initial amplitude of the wave. In the limit of small angles and

plane wave illumination this integral further can be simplified:

$$U(a,b) = B \int_A \int F(x-a, y-b)\, dxdy \qquad (2)$$

where*

$$F(x-a, y-b) = \exp\left\{\frac{ik}{2z}[(x-a)^2 + (y-b)^2]\right\}$$

The integral given in equation (2) can be extended by superposition to become

$$T * F = B \int_A \int T(x,y)F(x-a, y-b)\, dxdy \qquad (3)$$

where the region A is a collection of apertures each with transmittance described by the complex function T(x,y). The integral in equation (3) is a convolution (T*F) of the function T(x,y), the image, with the function F(x-a, y-b), the propagation function.

If one adds a plane reference wave, $A_r e^{ik\theta a}$, incident at a small angle θ to the hologram plane to the diffracted wave front, the intensity of the total wave front is given by:

$$\begin{aligned} H(a,b) &= |T * F(a,b) + A_r e^{ik\theta a}|^2 \\ &= |T * F(a,b)|^2 + A_r^2 + (T * F)A_r e^{-ik\theta a} \\ &\quad + (T * F)^* A_r e^{-ik\theta a} \end{aligned} \qquad (4)$$

In the reconstruction process, the hologram whose plate darkening corresponds to the function H(a,b) is illuminated by a plane, coherent, monochromatic wave. Again, there is a convolution, but now (H(a,b)* F(a-x, b-y)). The result of this convolution is three wave fronts: A wave with amplitude $|T * F(a,b)|^2 + A_r^2$ is propagated in the direction of the illuminating wave. A wave with amplitude (T*F) * F is propagated at an angle θ to the illuminating wave. This wave yields a real image, or a reconstruction, of T at a distance z. A wave front with amplitude (T*F)*F is propagated at an angle $-\theta$ to the illuminating wave. This wave is identical with that which would be observed from an object located a distance —z from the hologram, and thus yields a virtual image. Thus the first two terms in Eqn. 4 represent a central order image, the third term the real image and the fourth term the virtual image. The term $e^{-ik\theta a}$ acts as a shift operator and spatially separates the real image from the other two images; similarly the term $e^{-ik\theta}$ separates the virtual image from the other two.*

*We have recently modified this propagation function in order to spread the information over the whole of the hologram. This spreading is analogous to that produced by the diffuser plate used in optical holography. The numerical "diffuser" is achieved by multiplying F(x,y) by an appropriate real-valued function, e.g. 1-e $-c(x^2+y^2)$ for some constant c. Our technique does not increase bandwidth needed to describe the object whereas the optical diffuser does.

Computational techniques

The computer generation of holograms has been inhibited by the massive task involved in the straight forward numerical calculation of the convolution integrals of Eqn. 3. The first computer-generated holograms were reported by B. Brown and A. Lohmann[2] in 1966. These holograms were of the "binary mask" type which uses only two grey levels to encode the information in Eqn. 4. In 1966 J. Waters[8] extended this binary mask technique to three dimensions.

Huang and Prasada[6] suggested the use of transform techniques for holograms which, to first approximation, may be considered as Fourier transforms. In 1967, we[7] made holograms with 32 grey levels using an array of 64×128 for the object. At the time of this writing we have 10^5 resolution points in our image.

Since Eqn. 3 is a convolution integral, it may be readily evaluated using Fourier transform techniques. Indeed, the Fourier transform of H(a,b) is just the product of the Fourier transforms of T(x,y) and F(x-a, y-b):

$$H(\zeta, \eta) = T(\zeta, \eta)F(\zeta, \eta) \qquad (5)$$

The convolution can be computed by taking the inverse Fourier transform of $H(\zeta,\eta)$. In evaluating T * F digitally we use finite Fourier transforms[3] rather than infinite integrals.

A mathematical flow diagram of the program for hologram construction is given in Figure 1. A major part of the program is that in which T * F is evaluated using a fast finite Fourier transform subroutine. In our work T(x, y) is defined in basic "building blocks," arrays of 64×128 elements. These arrays are mapped onto the hologram plane using a propagation function F(x-a, y-b) which is defined over 128×512 elements. Conventional techniques for performing this map would require 2^{29} machine operations (by a machine operation, we mean a complex multiply and add). For an array of N elements, the fast finite Fourier transform technique requires only N $\log_2$ N operations; i.e., 2^{21} operations in our example.

In the two-dimensional case, a further simplification can be made. The propagation function F(x, y) is separable; i.e., $F(x, y) = F_1(x).F_2(y)$, where

*The first holograms, constructed by D. Gabor[5] in 1948 lacked the reference beam. He showed that the terms (T*F) and (T*F)* were present on his photographic plates, but he could not spatially separate them. A. Lohmann first proposed the two-beam technique described here. Independently, E. Leith and J. Upatnieks discovered the two-beam technique and made the first usable holograms with a monochromatic laser source.

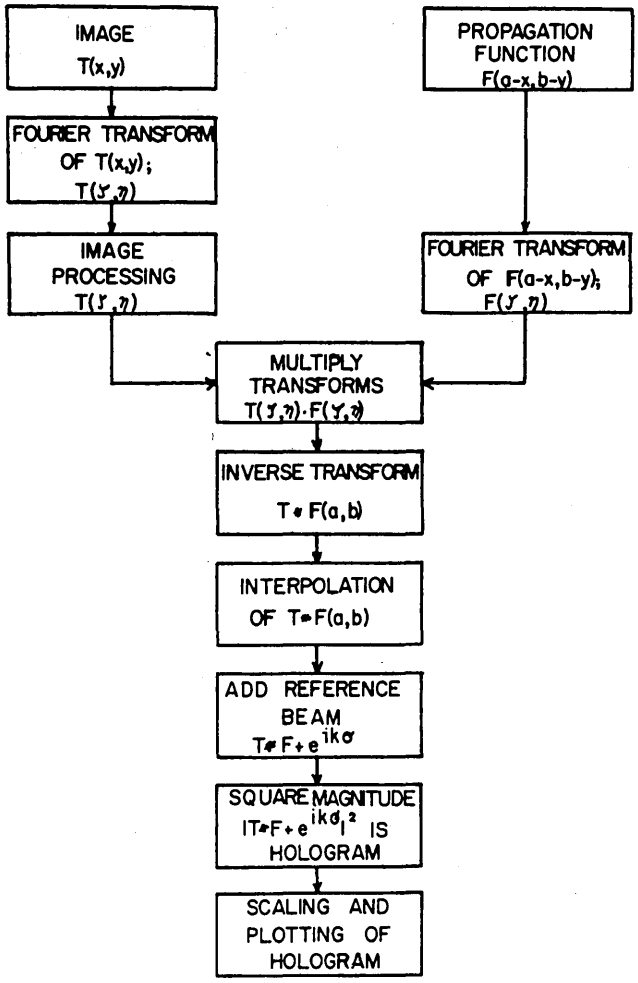


Figure 1—Flow diagram for hologram construction

$F_1(x) = e^{\frac{ikx^2}{2z}}$ and $F_2(y) = e^{\frac{iky^2}{2z}}$. Thus the Fourier transform of F satisfies $F(\zeta,\eta) = F_1(\zeta).F_2(\eta)$. If the propagation function is an NF x MF array, the savings in machine operations accomplished using the separability property is (NF.ML $\log_2$ NF.MF) — (NF $\log_2$ NF + MF $\log_2$ MF + NF.MF/2), a saving of 2^4 operations in our example.

These savings are not fully achieved however. The convolution which is obtained is not good over the NF x MF points of the propagation function, as can be demonstrated by the Helm-Sande theorem on finite convolutions.[4] If the object T(x,y) is defined over NF x MF points, where NF>NT and MF>MT, then the finite convolution of the two is written as

$$W(n,m) = \sum_{r=0}^{NF-1}\sum_{s=0}^{MF-1} B(r,s)F(n-r, m-s) \qquad (6)$$

$n=0,1,\ldots,NF-1$
$m=0,1,\ldots,MF-1$

where

$$B(r,s) = T(r,s) \quad \begin{matrix} r=0,1,\ldots NT-1 \\ s=0,1,\ldots MT-1 \end{matrix}$$
$$=0 \quad \begin{matrix} r=NT, NT+1, \ldots, NF-1 \\ s=MT, MT+1, \ldots, MF-1 \end{matrix}$$

The finite Fourier coefficients B(j, k) and F(j, k) are given by

$$B(j,k)=\sum_{r=0}^{NF-1}\sum_{s=0}^{MF-1} B(r,s)\exp\left(-2\pi i\left(\frac{rj}{NF}+\frac{sk}{MF}\right)\right)$$

and (7)

$$F(j,k)=\sum_{r=0}^{NF-1}\sum_{s=0}^{MF-1} F(r,s)\exp\left(-2\pi i\left(\frac{rj}{NF}+\frac{sk}{MF}\right)\right)$$

The inverse finite transform of the product B(j,k).F(j,k) is defined as

$$A(n,m)=\sum_{j=0}^{NF-1}\sum_{k=0}^{MF-1} B(j,k)F(j,k)\exp\left(2\pi i\left(\frac{jn}{NF}+\frac{km}{MF}\right)\right) \qquad (8)$$

The Helm-Sande theore mstates that W(n,m)=A(n,m) only for the region NT≦m≦NF−1 and MT≦p≦MF−1. Thus in general, W(n,m)≠A(n,m) for m<NT and p<MT, and the convolution obtained using the finite Fourier transform technique is valid only over (NF-NT) x (MF-MT) points. With our object defined in a basic block of 64 × 128 points and a propagation function defined over 128 and 512 points, we obtain a basic convolution block of 64 × 384 points. Our choice in the size of the basic block is dictated by the core available to us and convenience. We can increase the number of points in the convolution (the final hologram) by adding blocks.

A hologram requires resolution and frequency content four times greater than that needed to define the object. In the reconstruction process, only ¼ of the frequency content is used to construct the real image. The other ¾ is used to construct the virtual image and the central order image. One could compensate for this loss by decreasing the size by four and making a convolution four times larger. Of course, this would greatly increase the computing time. We have alternatively obtained good results by interpolating the convolution. We insert, by parabolic interpolation, three points between each pair computed in our convolution. Thus the interpolated convolution is made four times as large in negligible computer time. The interpolation requires approximately 3N operations compared to the N $\log_2$ N operations needed for the Fourier approach.

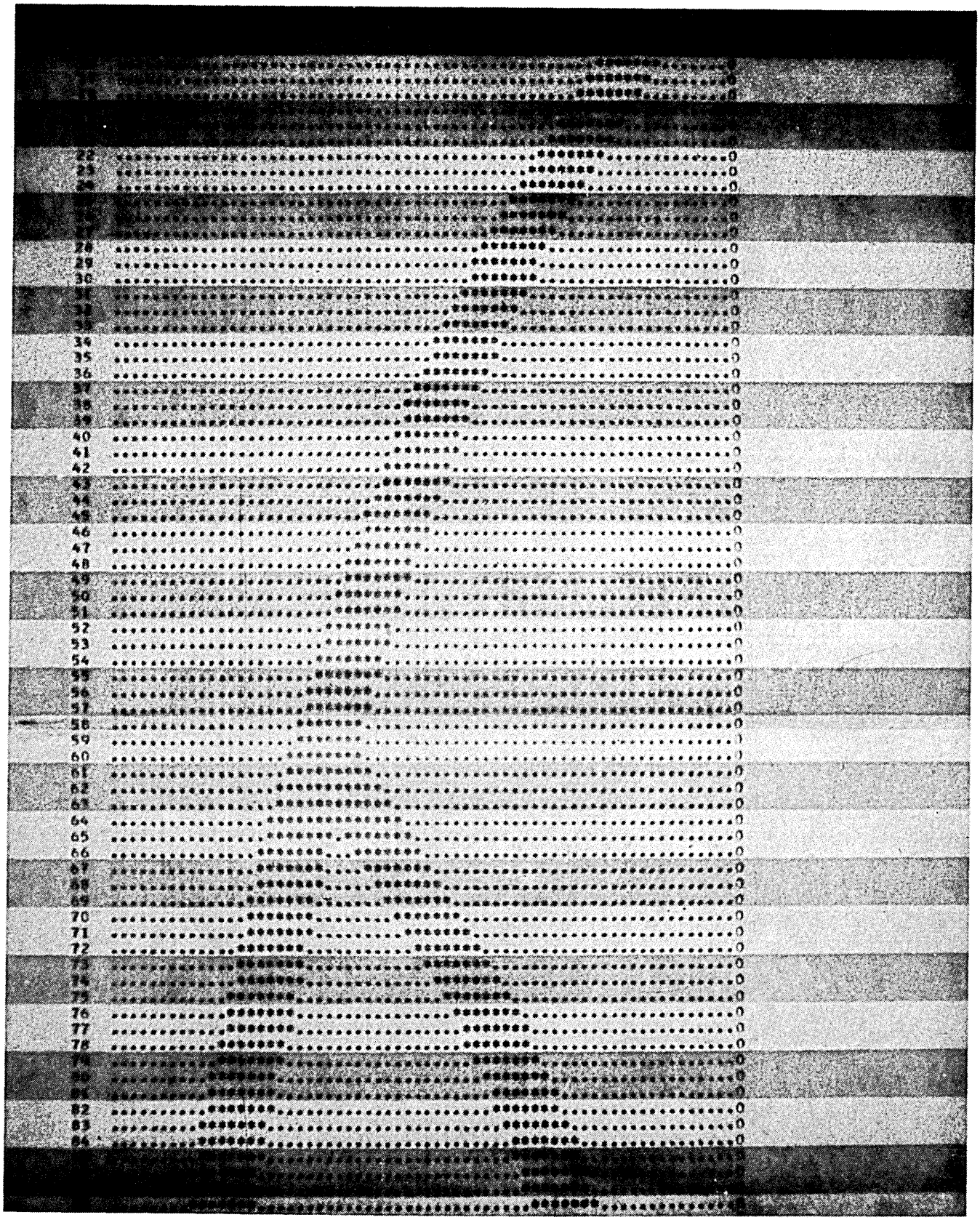


Figure 2—An image of the Greek letter lambda that was read into the computer

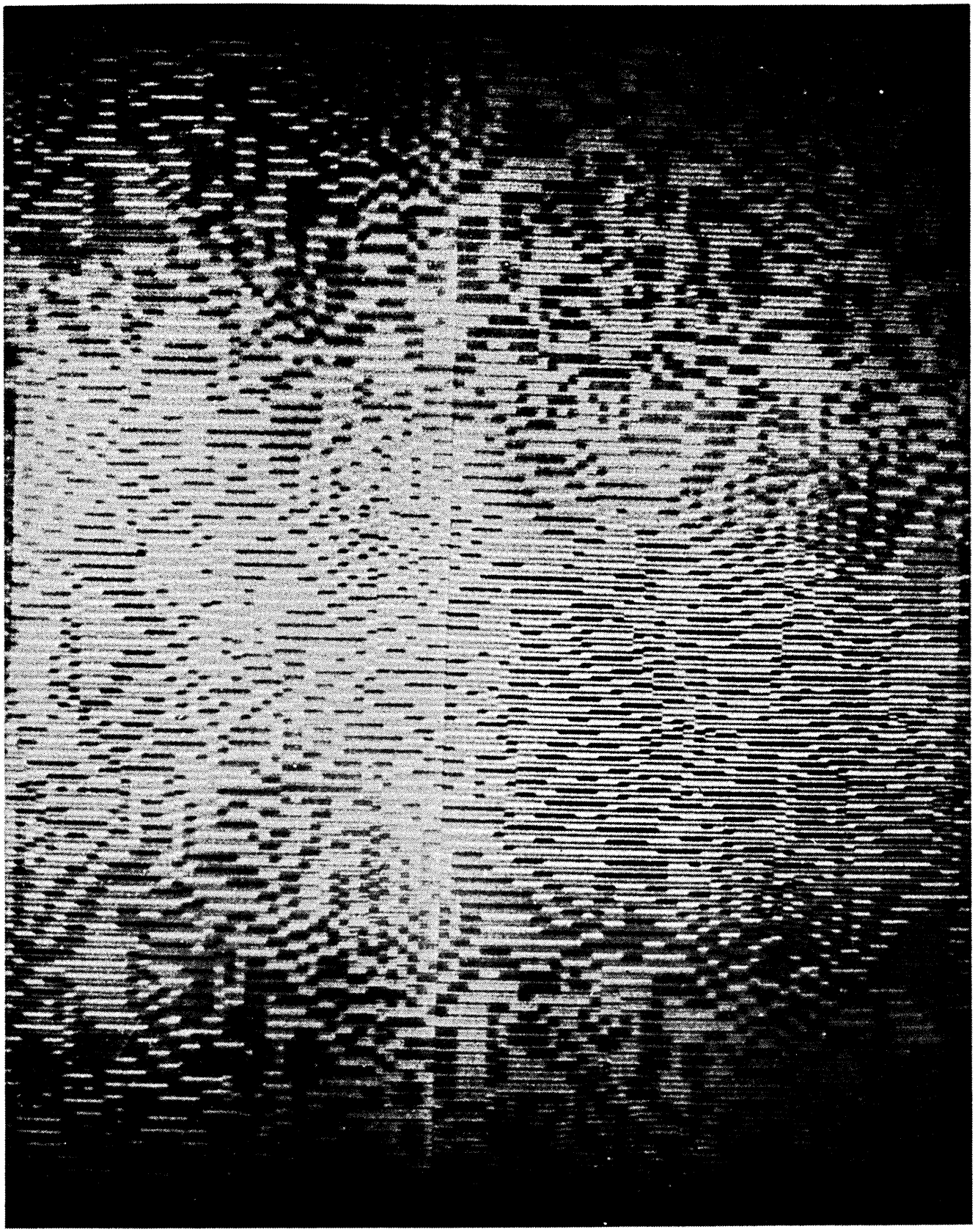
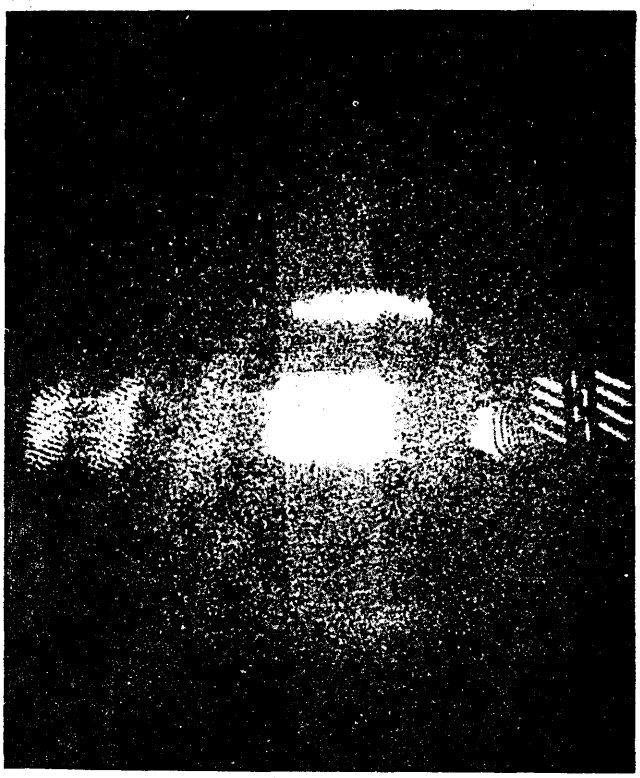


Figure 3—A digital hologram plotted on an IBM 2995, Model 2

Experimental verification of computer-generated holography

The test of a computer-generated hologram is whether the H(a,b) defined in Eqn. 4 can be translated to a photographic plate, which in turn can be illuminated by coherent light to form an image. An image is first digitally read in as the λ in Figure 2. We then compute H(a,b) and plot it using the IBM 2995 Model 2 photographic plotter, which has 32 grey levels and plots on a 40″ × 60″ piece of film with a spot size of 10 mils., Figure 3. We then photographically reduce this film to a 35 mm slide which is then illuminated with a collimated laser beam. Figures 4-6 exhibit the laser reconstructions which we have achieved from some of our computer-generated holograms. The image arrays contained 384 × 256 elements and the final hologram array contained 1530 × 256 elements. The physical dimensions of the reconstructed images are 1″ × 1.5″.



Note the virtual, central and real images

CONCLUSIONS

We have shown that the computer generation of holograms is feasible technique for the display of digitally defined data. Major savings in computer time can be achieved using the fast finite Fourier transform algorithm and by taking advantage of the separability of the propagation function. Further improvements can be achieved using interpolation.

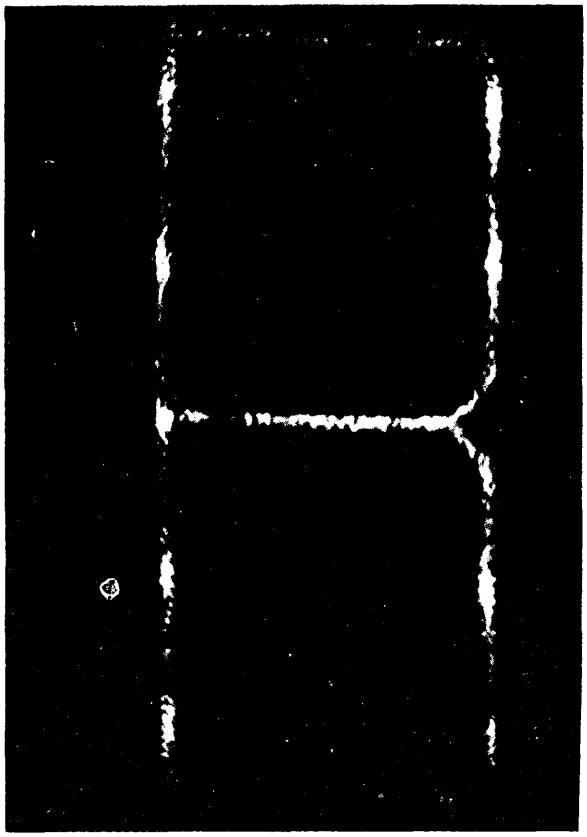


Figure 5—The real image of an optical reconstruction of the letter "B"

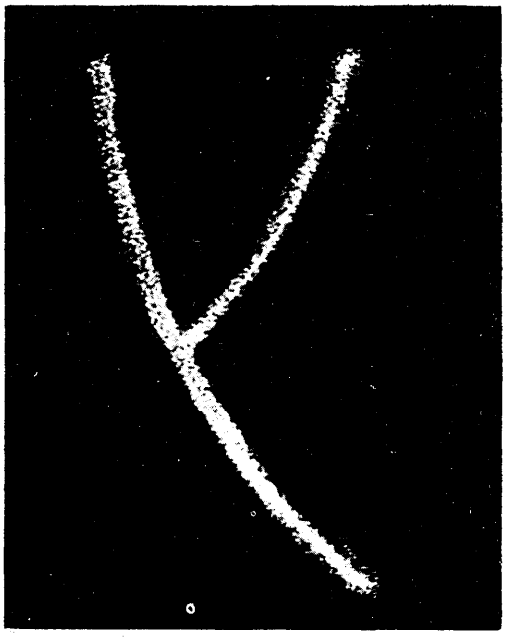


Figure 6—The real image of an optical reconstruction of the lambda in Figure 2

ACKNOWLEDGMENT

We wish to acknowledge the efforts of Mr. D. E.

Richards who did the bulk of the programming for the computations reported herein. His work in producing efficient programs has greatly expedited our work.

REFERENCES

1 M BORN E WOLF
Principles of optics
Pergamon Press New York 1959

2 B R BROWN A W LOHMANN
Complex spatial filtering with binary masks
Appl Optics vol 5 p 967 1966

3 J W COOLEY J W TUKEY
An algorithm for the machine calculation of complex Fourier series
Mathematics of Computation vol 19 p 297 1965

4 H D HELMS
Fast convolutions using the fast Fourier transform
Unpublished Technical Memorandum Bell Telephone Laboratories Inc 1966

5 D GABOR
Microscopy by reconstructed wavefronts
Proc Roy Soc vol A 197 p 454 1949

6 T S HUANG B PRASADA
Considerations on the generation and processing of holograms by digital computers
MIT Res Lab of Elect Quar Prog Rep no 81 p 199 1966

7 L B LESEM P M HIRSCH J A JORDAN JR
Computer generation and reconstruction of holograms
Proceedings Polytechnic Institute of Brooklyn Symposium on Modern Optics 1967

8 J P WATERS
Holographic image synthesis utilizing theoretical methods
Applied Physics Letters vol 9 no. 11 pp 405-407 1966

Half-tone perspective drawings by computer

by CHRIS WYLIE, GORDON ROMNEY, DAVID EVANS
and ALAN ERDAHL
University of Utah
Salt Lake City, Utah

INTRODUCTION

In recent years, the sheer increase in demand for the graphic presentation of three-dimensional objects has almost overhwelmed conventional facilities; that is, designers, draftsmen and especially engineering artists. For example, it is important for a designer or architect to quickly describe a three-dimensional object and view it immediately; not as an endless set of engineering drawings, but as if he were viewing the three-dimensional object itself. He should be able to take a distant look at a complicated object, and then view, in detail, any subsection of the object. In other words, he would like to quickly and cheaply simulate and view the thing he is designing.

The goal of this project is to provide a system which will display images that a person can "feel," as contrasted with images that he must laboriously interpret (e.g., the engineering drawings of an airplane).

Several subjective factors apparently help the viewer's ability to "feel" the overall structure of a three-dimensional object: 1) binocular (or stereo) vision, 2) elimination of the hidden surfaces, 3) recognition of distance and shape as a function of illumination (or shading), and 4) real time movement.

For a display algorithm to be practical, the computing time should grow only linearly with the complexity of the object and the resolution of the display. Other workers* have found that, with their methods for the hidden surface problem, the computing time grew very rapidly with complexity. Thus, the display of significant objects was thus impractical.

There were other disadvantages. Roberts used rectangular solids and prisms to construct objects. This is a severe limitation when dealing with curved or Riemannian surfaces. To get around this difficulty, we have used triangles to describe objects. For example, it is easily seen that it is impossible to completely cover the surface of a sphere with quadrangles. However, it can be done quite conveniently with triangles (Figure 1).

Any developable surface can be approximated arbitrarily accurately with small, but finite, triangles (Figure 1). Another reason for using traingles is that three points always determine a plane. In this case many results from geometry and linear algebra have attractive forms for computation.

The object, and its perspective projection on a view plane, are examined by a scanning ray extending from a view point (Figure 2).

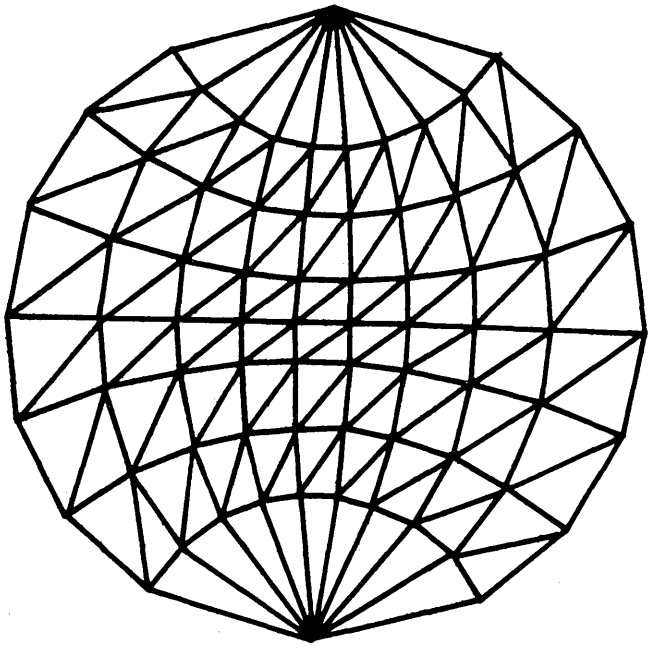


Figure 1 – Example of one method of approximating a sphere by planar triangles

*Lawrence G. Roberts, Lincoln Laboratory, Massachusetts Institute of Technology.

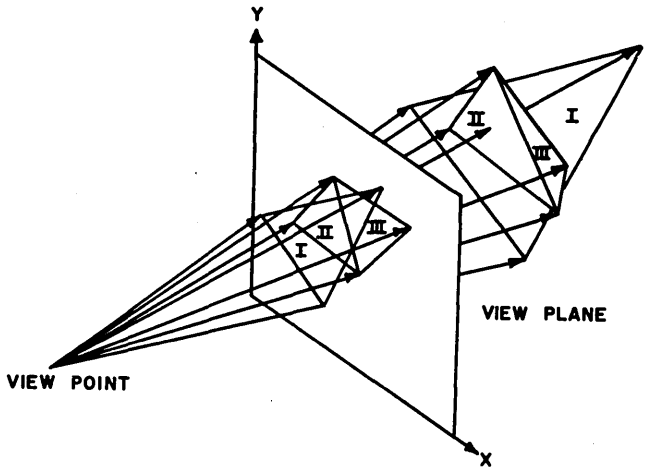


Figure 2 – Perspective projection of triangles onto the view plane

Overview of the algorithm

I. Background Concepts

A. Basic geometry of the problem

Everything is ultimately referred to an underlying, orthonormal vector basis $\vec{E}_i$ (Figure 3). The object is viewed from an arbitrary vantage point specified by $\vec{P}$. The viewing plane is parallel to vectors $\vec{e}_1$ and $\vec{e}_2$. The angular orientation of the viewer about the $\vec{e}_3$ axis is determined by $\vec{e}_1$ and e_2. Every three-dimensional triangle determines a two-dimensional perspective image on the view plane. In Figure 2, triangles II and III are in front of triangle I.

B. Illumination of the object

The present algorithm allows only a point source of illumination at the view point (like a single flashbulb photograph). As a consequence, there will be no shadows in the picture. The apparent brightness of a point on a surface depends on the following:

1. The basic physical laws governing incident light, e.g., light flux varies as the inverse square of the distance from a point source, and the amount of light incident on a surface is a function of the angle of incidence (the angle between the incident ray and the normal to the surface).
2. The nature of the reflecting surfaces, e.g., the reflectivity, texture and color may vary.

We have arbitrarily chosen an inverse fourth law for computational simplicity. The user, however, may employ any relationship he desires, by modifying the appropriate subroutine.

C. Hidden surfaces problem

By far the major obstacle is solving the hidden surface problem and the means of preventing the computing time from growing faster than the number of triangles. Most of this paper will be devoted to this problem.

In solving the hidden surface problem, one could compare all the components of the entire surface for each point in the picture. This leads to a computation which likely grows at least as the product of the resolution and the number of surface elements. Instead, by using special sorting algorithms, only those triangles intersected by the scanning ray (Figure 2) need be compared.

II. Algorithm

This is a greatly simplified version to avoid getting bogged down in programming details.

A. We have already assumed that any object may be approximated by a set of triangles. The input data is a set of arbitrarily ordered triangles specified by the three-dimensional coordinates of their vertices (Figure 3).

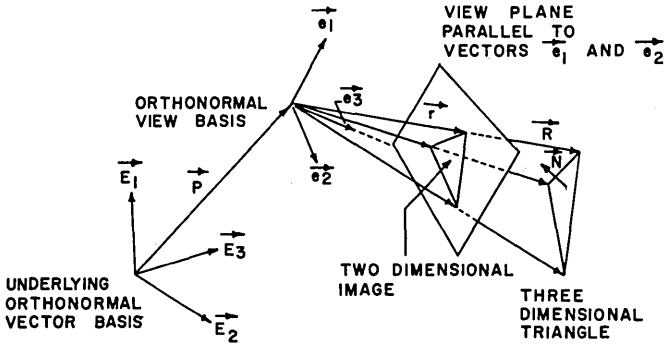


Figure 3 – Basic geometrical convention for the algorithm. The unit normal to a triangle is $\vec{N}$. The coordinate variables are (x^1,x^2,x^3) for the underlying basis $\vec{E}_i$, and (x^1,x^2,x^3) for the view basis $\vec{e}_i$.

The following must be specified (Figure 3):

1. View point, $\vec{P}$.
2. View basis, $\vec{e}_i$.
3. Distance from the view point to the view plane.

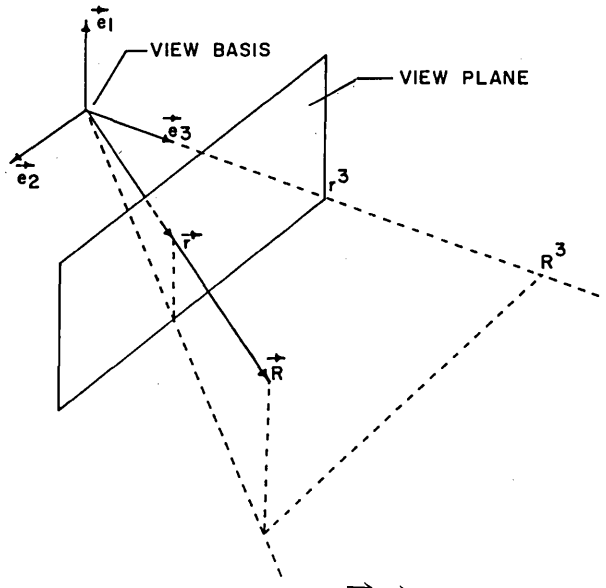


Figure 4 – Illustration of the ratio $|\vec{R}|/|\vec{r}$ being equal to R^3/r^3

B. Per frame calculations.

The view plane is examined by a systematic scanning raster (Figure 5). One complete raster scan of the view plane will be called a frame.*

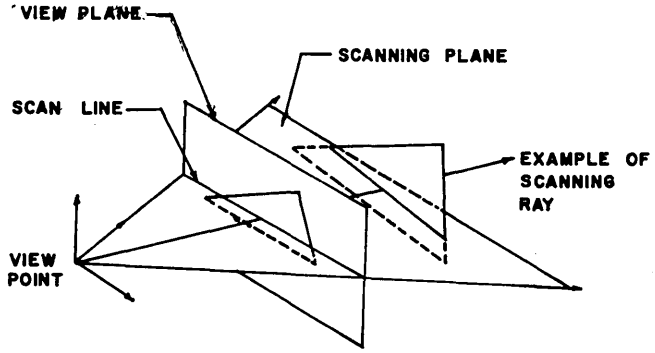


Figure 5 – Slice made by the sweep of a scanning ray

1. Preprocess the triangles.

The algorithm begins with the calculation of various quantities about each triangle that will be needed for later computations. They are not, however, essential in order to understand the basic method.

a. Find the normal $\vec{N}$, to each object triangle. This is needed in the apparent brightness calculation (Figure 3). From Figure 3, it is seen that the unit normal $\vec{N}$, to the three-space triangle may be calculated by normalizing the vector cross product of any two of the triangle's sides.

b. Find the apparent brightness at each object triangle's vertices, assuming the source of illumination is at the view point.

1) To find these brightnesses, the distances from the view (or illumination) point to each of the three vertices of the triangle must be determined.

It is assumed that we have either been given (or have transformed) the components of all three-space position vectors so they are relative to the view basis.

2) The magnitude of a position vector to a vertex is:

$$|\vec{R}| = \sqrt{(R^1)^2 + (R^2)^2 + (R^3)^2}$$

Given the unit normal of each triangle and the distance to each vertice, the user may calculate the apparent illumination of the triangle vertices with any formula he desires. The formula chosen will generally vary primarily with the nature of the reflective surface.

c. Calculate the linear brightness interpolation parameters as in Section II, E,2.

Essentially, we have assumed that the apparent brightness in the interior of the view plane image of a triangle is adequately approximated by linear interpolation of the brightnesses at the three vertices of the triang'e. The formula is easily derivable and will not be discussed here.

d. Calculate the linear distance ratio parameters.

1) The distance ratio w along any particular scan ray is defined to be:

$$w = \frac{\text{(Distance from view point to object)}}{\text{(Distance from view point to image)}}$$

In Figure 3, $w = \dfrac{|\vec{R}|}{|\vec{r}|}$ for the scan ray R to the uppermost vertex. The distance ratio is calculated for all three vertices of each triangle. For a given view point, three linear distance ratio parameters, a,b, and

*One side benefit of the raster scan is the inherent compatibility of the method with television-type display devices.

c, are then computed* for each triangle such that distance ratio w, is:

$$w = ax + by + c$$

for any scanning ray intersecting a given triangle at view plane coordinates x and y.

2) The calculation of a, b, and c is based upon the fact that the distance ratio w, as defined above, is also given by:

$$w = \frac{[\vec{e_3} \text{ component of vector to object}]}{[\vec{e_3} \text{ component of vector to image}]} = \frac{R^3}{r^3}$$

The component r^3, to the image, is simply the distance from the view point to the view plane, which is constant throughout a frame calculation (Figure 4).

The component R^3, to the object, is found directly from the equation of the plane which is determined by the three vertices of the three-space triangle. Note that the view plane coordinates (x,y), are effectively the same as (x^1,x^2), defined by the view point basis $(\vec{e_1},\vec{e_2},\vec{e_3})$. Because the view plane is two-dimensional and the view basis three-dimensional, there is no direct correspondence of the x^3 coordinate to any view plane coordinate. The equation of the plane coincident with the three-space triangle is:

$$x^3 = fx^1 + gx^2 + h$$

where f, g, and h are constants describing the plane of the object triangle and x^1 and x^2 are equivalent to the view plane coordinates x and y, of its image.

Now the distance ratio is expressible in terms of the position of the scanning ray on the view plane. This is from the fact that from the projective properties of the situation (Figure 4), the distance ratio w is:*

$$w = \frac{|\vec{R}|}{|\vec{r}|} = \frac{R^3}{r^3}$$

Components R^3 and r^3 have been determined in the discussion above.

2. Project all three-dimensional triangles onto the view plane to make a set of two-dimensional triangles (Figures 2 and 3).
3. Because the projected image is going to be scanned from top to bottom, a line at a time, all the triangles are sorted with respect to y to eliminate from consideration those triangles not crossed by the current scan line.
   (Figure 6).
   a. Sort the three vertices of each triangle with respect to y.**
   b. The y-entrance table (Figure 7), tells which triangles are entered (i.e., begin to be intersected) by the scan line at a given y.
   c. The y-exit table, which is identical in structure to Figure 7, tells which triangles are exited by the scan line at a given y.
   d. This way we will only have to look at those triangles that a given scan line actually crosses. This avoids examining all the triangles in the entire picture at each scan line (Figure 6).
4. Start the y-scan
   Every time the scan is incremented by y, the y-occupied table (Figure 8) is updated, if necessary, by looking at the y-entry and exit tables to see if a triangle has been entered or exited. In Figure 6, the scan line moves downward and triangles are entered and exited. We turn on an occupied flag when a triangle is entered and turn it off when it is exited. Each triangle has its own location in

*These are recomputed when and only when the view point and orientation are changed.

*In programming this algorithm, it should be noted that various combinations of triangles present cases that require arithmetic of great precision: e.g., triangles with common edges or vertices; triangles that are coplanar, or very nearly so. These cases are not discussed here, but should be carefully examined by the programmer. From our experience, we have concluded that the description of the object and all computation should be done in integer arithmetic. Also there should be a total avoidance of division to eliminate roundoff error that would make the logic of special cases ambiguous.

**This sort actually switches the vertex data in core storage.

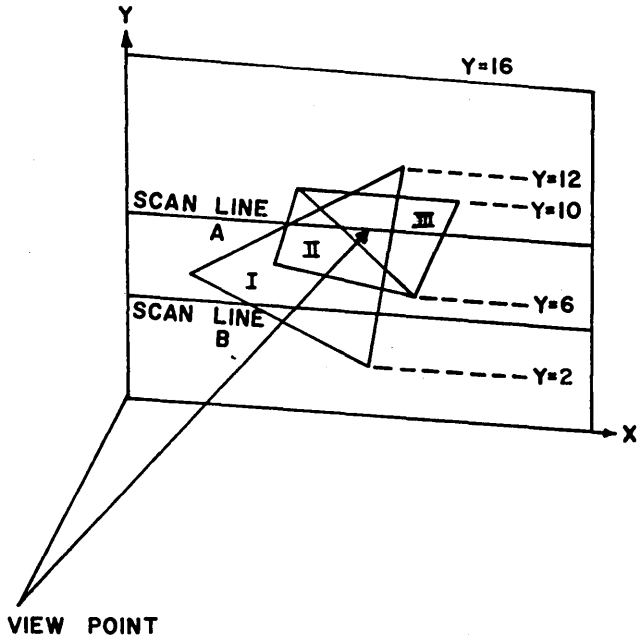


Figure 6—Scan line intersections of projected triangles on a simplified view plane

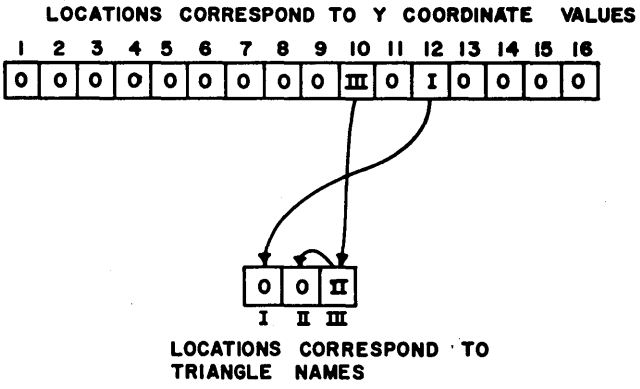
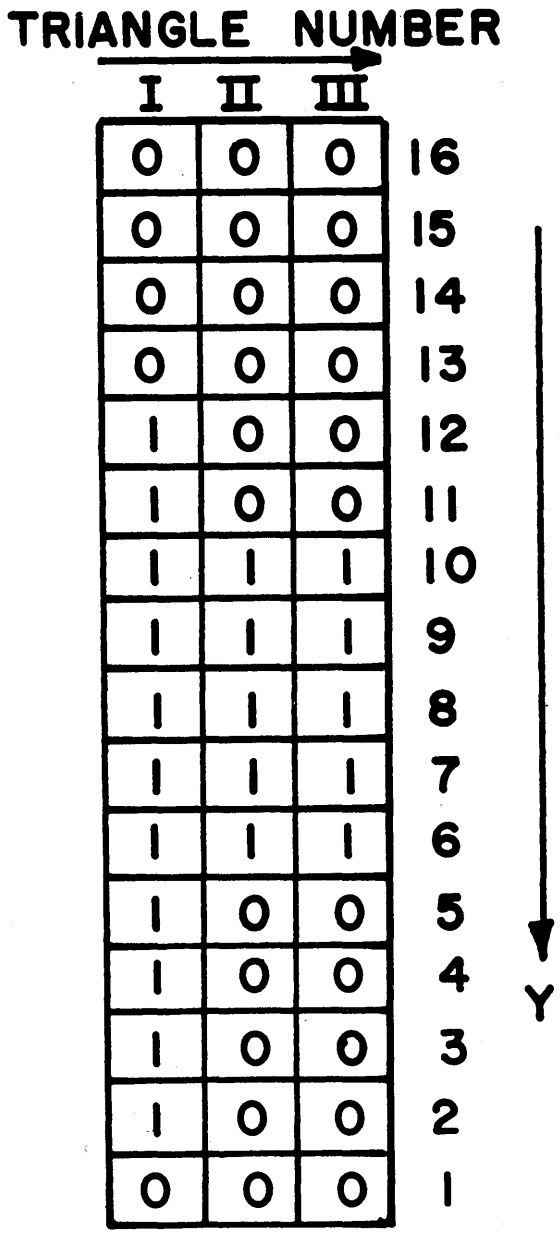


Figure 7—The y-entry tables corresponding to the triangles in Figure 6. This is essentially a pointer sort indicating which triangles enter at a given y.



QUENCE OF Y-OCCUPIED BLES

Figure 8—For a given y scan, an occupied table indicates those triangles the scan line intersects. This figure shows a sequence of occupied tables for the entire segment of the view plane illustrated in Figure 6. Only a single row of the occupied table will exist at any instant for the current y

which to put an occupied flag (Figure 8). This is the first major reduction in computation.

C. Per scan line computations
   1. For the current scan line:
      a. We go to the y-occupied table and get only the triangles that this scan line crosses.
      b. Find the x values of the intersections of the scan line with the sides of the view plane images of the triangles pulled out of the y-occupied table (Figures 6 and 9).
   2. Sort each triangle's x-entry and exit intercepts into the x-entry and exit tables

respectively. The x tables and sorts are identical to those used in the y-entry and exit sorts shown in Figure 7. They will not be shown in a figure.

3. Commence moving the scan ray along the scan line by x increments.

D. Per point calculations

1. For the current x, look at the x-entry and exit tables to see if there is an intersection at this x. When an intersection point is encountered, we want to know if there is a change in the visible (or hidden) status of a triangle.
   a. If there is an intersection
      1) Update the x-occupied table. The only time a triangle can change its visible or hidden status is at an intersection of the scan line and a triangle side (Figure 9).*

   All these intersections are calculated for for the current scan line and are used, via manipulation of the x-entry and exit tables, to update the x-occupied table.**

   It is at this stage that a triangle is either added or deleted from the x-occupied table.

   2) Then go into the hidden parts calculation. (Section II,D,2)
   b. If there is no intersection, increment x and test for an intersection at the new x (i.e. increment x and go to Section II,D,1).
2. Hidden parts calculation
   a. Examine all triangles in the x-occupied table for the current x.
   b. Using the distance ratio parameters (computed previously in the per frame calculations Section II, B, 1, d), calculate the distance along the scan ray from the view point to each of the three-space triangles entered in the x-occupied table (Figure 9).
   c. Sort the distances to find the smallest. The triangle with the smallest distance ratio is the visible one and must be added to the visible table.

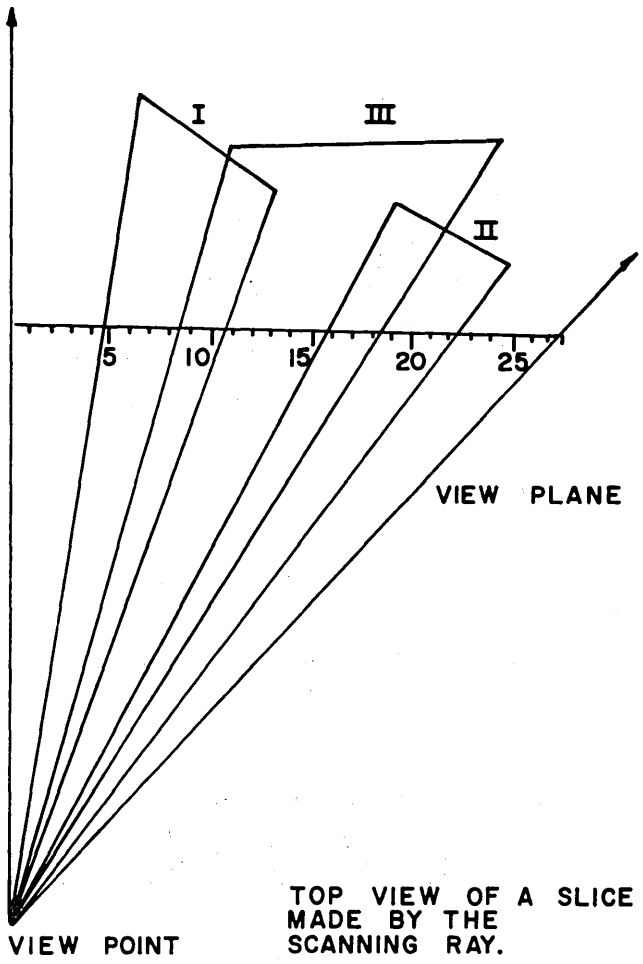


Figure 9—Top view of a slice made by the scanning ray

LOCATIONS CORRESPOND TO THE VALUE OF X

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0	0	0	I	I	I	I	I	I	I	III	III	III	III	II	II	II	II	II	II	II	0	0	0

Figure 10—The visible table contains the triangle number visible at the designated x position of the scan ray.

3. Add to the visible table
   a. Figure 10 shows a completed visible table for a scan line.
   b. The contents of the table are triangle names (or numbers) showing which triangle is visible at a given x.
   c. For each x increment of the scan ray update the x-occupied table. Using the x-occupied table and the distance computation, the visible table is modified by placing the name of the visible

*We assume that no triangles cross through any other triangles. Intersecting triangles may be resolved into non-intersecting triangles by a separate algorithm prior to execution of the present half-tone perspective algorithm.

**The x-occupied table has the same structure and is built in the same way as the y-occupied table. Consequently it will not be shown.

triangle in the location corresponding to the present x value (Figure 10).

d. The visible table is constructed for an entire scan line and is then used to find which triangle's intensity interpolation parameters are to be used for each x.

E. Per point intensity calculations

Calculate the intensities for each x in the current scan line.

1. We interpolate to find the intensity over the visible interior of a triangle using only the intensity values at the three vertices. This allows a considerable reduction in computing time. For simplicity and speed, but not necessity, we chose linear interpolation. The linear interpolation parameters have already been calculated and stored during the per frame calculations (Section II, B, 1, c).
2. The formula for the intensity at a point x on the scan line y is

$$I = ax + by + c$$

Where I is the intensity and a,b, c are the linear interpolation parameters for the visible triangle.

F. Output to display device

The list of intensities for this scan line is sent to a peripheral device for eventual display. The output subroutines are distinct and independent of the half-tone algorithm to permit flexibility as the display hardware is improved or altered.

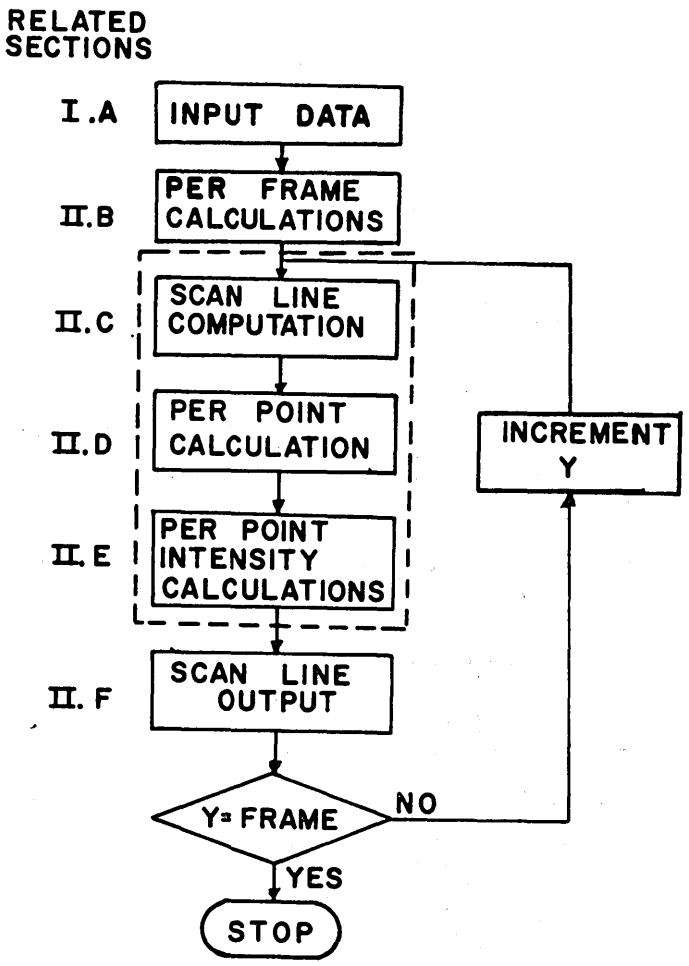


Figure 11 – Flow chart of the half-tone perspective algorithm

Results

I. Program

A FORTRAN IV program of the algorithm (Figure 11), called PIXURE has been written* and used to produce half-tone pictures of a cube and tetrahedron (Figures 13 through 18). For both the cube (12 triangles) and the tetrahedron (4 triangles) the execution time of PIXURE was roughly 25 seconds to calculate a frame of 512 x 512 points on a Univac 1108. PIXURE at present is approximately 3800 Univac 1108 assembly language instructions in length and occupies 14K 36-bit words of storage for a picture of 100 triangle complexity.

Preliminary tests indicate that the execution time is most dependent on the number of scan lines that intersect the two-space image of the object (e.g. there are eleven scan lines, $2 \leq y \leq 12$, that intersect triangles in Figure 6). It also appears that this dependence is very closely linear. On the other hand, execution-time dependence on the number of triangles (i.e., the number of intersection points per scan line) appears to be much better than linear. The dependence on the number of hidden triangles per intersection point has not been rigorously determined, but seems to be close to linear.

II. Hardware techniques

Each scan line that PIXURE generates is sent to a PDP-8 via a specially designed interface (Figure 12). The PDP-8 serves essentially as 1) a buffer, 2) a raster generating device for an oscilloscope, and 3) an a-synchronous I/O channel communicating with the 1108. Each scan line, in turn, is stored in the PDP-8 memory and then transmitted through a digital to analog (D-A) converter to a Tektronix 453 oscilloscope. The scan position is dictated by ten bit x and y registers in the D-A converter. The intensity of the beam at each point in the scan is controlled by a six bit z register. Due to storage and 1108 – PDP-8 transmis-

*Modifications of this program and some new work are progressing rapidly.

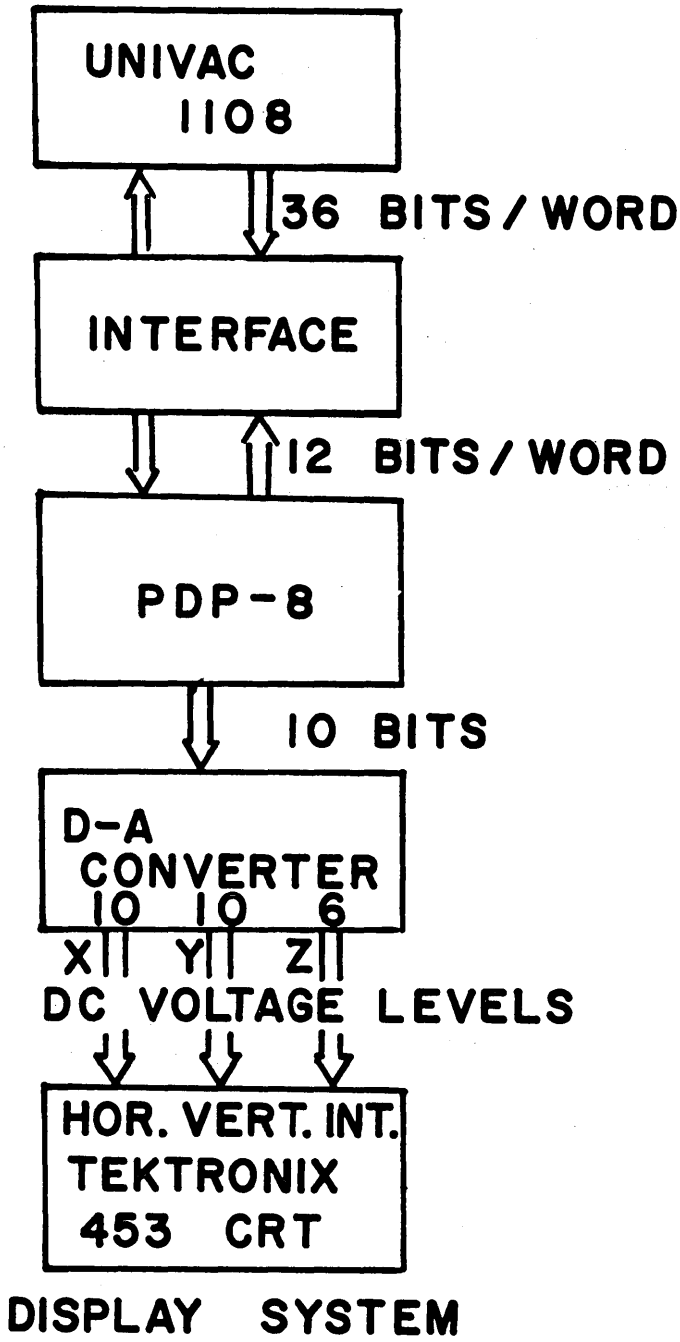


Figure 12 – Display system

sion-rate limitations we have been forced to take time exposure photographs of the scope trace. As soon as a scan line is completed, the PDP-8 requests information for the next scan line. For a 512 x 512 frame it takes approximately ten seconds to generate a picture.

III. Subjective interpretations

The principal objective of this project is to allow people to see three-dimensional objects, as realistically as possible, using two-dimensional images (or displays). We have chosen to erase hidden surfaces and use half-tone shading to give the illusion of depth (or distance) and indicate spatial relationships. Although we are presently limited to a single source of illumination at the view point; nonetheless, the pictures of our test objects show obvious dimensionality.

Figures 13, 14 and 15 represent a cube whose resolution differs by a factor of ten. It is evident that Figure 14, representing a picture of 512 x 512 points,

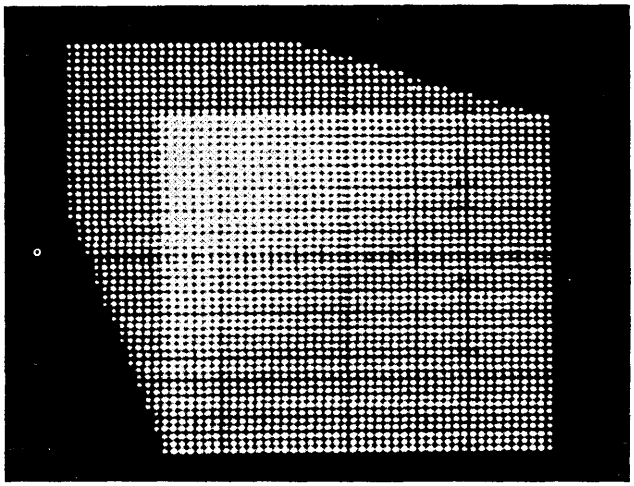


Figure 13 – Cube 100 × 100

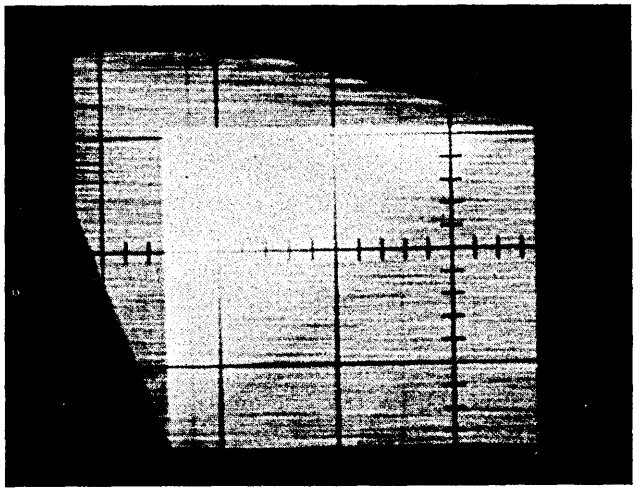


Figure 14 – Cube 512 × 512

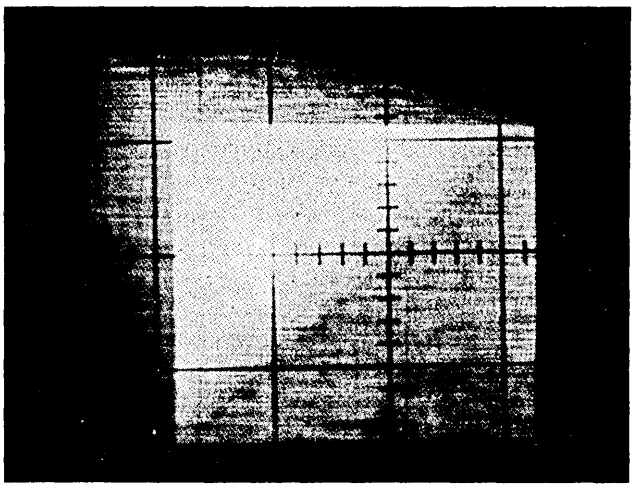


Figure 15—Cube 1024 × 1024

supplies sufficient information to adequately describe the cube. A more critical test on the resolution of the receding edge could not have been made, and yet, the edge appears tn the higher resolution pictures. The unusual perspective, however, was merely the result of an arbitrary choice in geometry. The apparent triangular composition of the cube faces has since been corrected and a smooth transition across triangle boundaries achieved (Figure 16).

The pictures of the tetrahedron (Figures 17 and 18) are superior in quality to those of the cube for two reasons. First, a defect in the display hardware was partially corrected, resulting in a more even display pattern. Scan lines are still noticeable, but it is felt that additional improvement in the hardware will significantly diminish this defect. The second improvement was in the selection of a more correct range of intensity levels used in the brightness calculation.

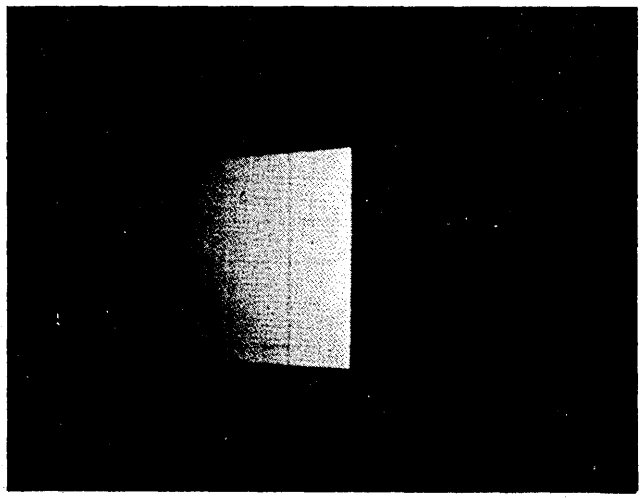


Figure 16—Cube 512 × 512

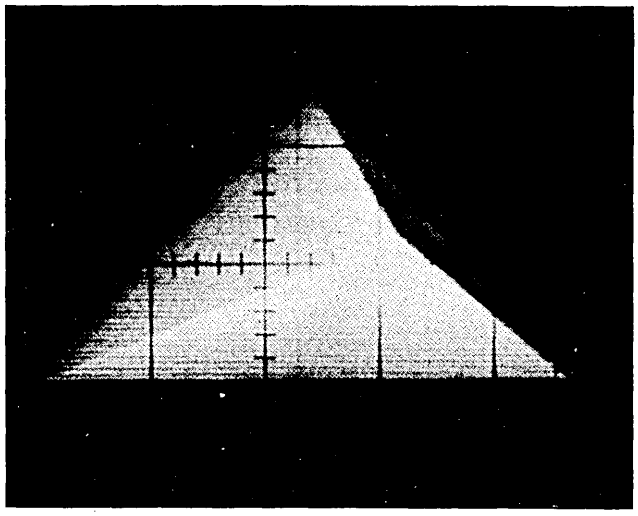


Figure 17—Tetrahedron 512 × 512

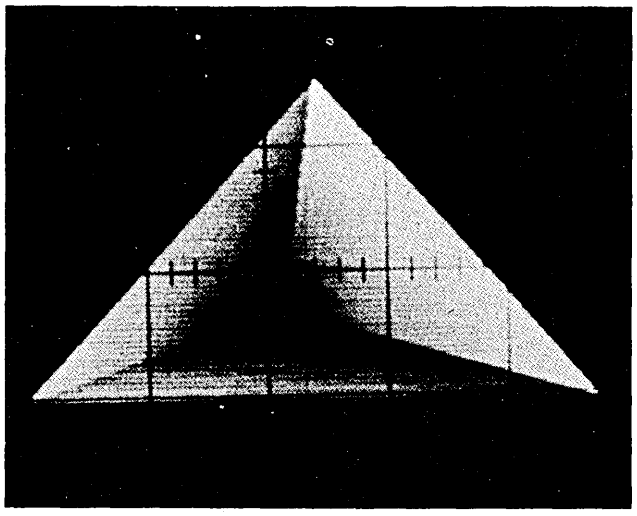


Figure 18—Concave tetrahedron 512 × 512 (one side removed)

Another objective is to display an object so that it will not be ambiguously interpreted. The tetrahedron in Figure 17 is decidedly convex, but Figure 18 could be either convex or concave unless the source of illumination be specified.

SUMMARY

In the cases we have tested, the computing time grows almost linearly with the resolution of the picture, the size of the visible portion of the object and apparently, the amount of hidden surface. This makes the algorithm practical, and is a result of special

sorting techniques which greatly reduce the number of hidden surface comparisons required. The objects we have displayed appear quite three-dimensional and their hidden surfaces are effectively eliminated. The computing time required for a picture composed of over 10^6 points was approximately 40 seconds on a Univac 1108.

The present system definitely proves the feasibility of the real-time display of two-dimensional half-tone images. It is felt that the technique may be easily extended to stereo representation of half-tone images. Furthermore, the algorithm is so constructed as to allow computations to be executed in parallel (see the dotted section in Figure 11). As many scan lines as hardware permits may be calculated simultaneously. Also, much of the computation may be performed by incremental hardware. The parallel and incremental characteristics of the algorithm lead us to believe that real-time movement and display of half-tone images is very near realization.

A typical user wishes to describe an object in a form convenient for him. Also, a flexible and extensive data structure must be constructed to contain and manipulate an object. Therefore, the practical application of the algorithm depends greatly on the ability of the system to convert an object into a suitable mesh of triangles. Our group has initiated work in these directions and at present has a triangle generation algorithm operational for objects composed of planar surfaces.

ACKNOWLEDGMENTS

The authors are deeply indebted for the programming assistance of Lee Copeland and Richard Blackburn. The technical skills of Richard Jepperson, Charles Eder and Y. T. Kim have assisted immensely in helping produce the first photographs. And, last but not least, we wish to thank the University of Utah Computer Center for their patience and assistance in making these results possible.

BIBLIOGRAPHY

A L FASS and A R AMIR-MOÉZ
Elements of linear spaces
Macmillan Company New York 1962
B E MESERVE
Fundamental concepts of geometry
Addison-Wesley Reading Mass 1955
L G ROBERTS
Homogenous matrix representation of N-dimensional solids
MIT Lincoln Laboratory Lexington Mass
L G ROBERTS
Machine perception of three-dimensional solids
MIT Lincoln Laboratory 1963 Technical Report no 315 Lexington Mass

VISTA–Computed motion pictures for space research

by GEORGE A. CHAPMAN
Computer Sciences Corporation
Silver Spring, Maryland

and

JOHN J. QUANN
National Aeronautics and Space Administration
Goddard Space Flight Center
Greenbelt, Maryland

INTRODUCTION

The application of digital computers to reduction of telemetry data assumes an increasingly important role in the analysis of physical problems.[1] Even after reduction of the raw sensor data, an immense volume of resultant data remains. Thus there is a constant need of new tools for computer-aided analysis, synthesis and display of scientific results.

Graphic aids have evolved from printer-listings, through lineprinter plots to a wide range of x-y plotter techniques. Cathode Ray Tube devices (hard copy and one-line console) have introduced the latest generation of display capability. The trend, with this latest capability, is from static x-y plots to dynamic time-sequence displays.

This paper presents a generalized technique for generation of off-line CRT motion-picture displays of objects in three-dimensional space. Initial implementation of the system, called VISTA* (Visual Information for Satellite Telemetry Analysis), uses the Univac 1108 computer to generate time-sequence displays for output on the Stromberg Carlson 4020 microfilm plotter. The system accepts satellite orbit and attitude data and produces motion pictures illustrating the three dimensional position and orientation of the spacecraft in orbit, relative to one or more celestial bodies.

VISTA—General description

VISTA is a generalized system which accepts spacecraft orbit and attitude information and creates a physical picture of that data. The picture is not what is normally referred to as a data plot but rather a true representation of the spacecraft in orbit about a central body. The program output is synchronized with orbit time and is used to prepare multiple frames of 16 mm or 35 mm film on the SC-4020.

Current statistics indicate that one to two 1108 computer seconds is required to produce each individual frame (as illustrated in Figure 2). This variation in computer time is due to change occultation requirements. Computing time approaches a maximum when one body occults a dense portion of the earth's shoreline. At a camera rate of 16 frames per second, one minute of viewing time requires 24 minutes of 1108 computer time.

When the film is viewed through a projector as a motion picture, a continuous time history of the motion and orientation of the spacecraft is observed thus enabling a viewer to assimilate in a few seconds a large volume of data while subjecting the data to a rapid and comprehensive analysis.

There exist two major subsystems in VISTA over which the user has complete control and the potential for dynamic alteration:

1. Presentation of a mapped earth system and a spacecraft body system, ranging from a simple point-mass display to an orthographic projection of a solid spacecraft, which can be viewed from any arbitrarily chosen vantage point.
2. Presentation of data from an experiment aboard the spacecraft in one or more of several forms designed to aid the viewer in his analysis of the data.

*VISTA was developed by Computer Sciences Corporation for NASA-GSFC.

The first subsystem is concerned with the dynamics and the orthographic projection of bodies in motion in some arbitrarity specified coordinate system. In the case of three-dimensional objects, construction lines, object lines and detailed markings not visible to the viewer are automatically deleted from projection. This "hidden line" capability is used when occultation occurs between one or more of the included bodies. The user of the VISTA system has complete control over the desired view angle or vantage point. The viewers location and view vector may be either static or dynamic.

The second subsystem is primarily concerned with presenting and formatting data, title frames and selected overlay information. In this area, the prime objective was to overcome the inherent jitter associated with data plotting in a motion picture atmosphere and at the same time include complete data presentation capabilities. Both objectives have been successfully achieved and a wide variety of options and presentation formats are included to aid the viewer in his analysis of the data. These capabilities include presentation of data in the form of a "clock" with the hands indicating orbit sensor measurements. The speed at which the different hands move reflects relative speed and acceleration of the data variables. Two and three dimensional grids are also available where the latter may be rotated to a "best" view. Either the data or the grid can be oriented for rapid recognition as the frames are repeatedly updated and projected. A "marker" is also available, which points to the specific data point associated with the given display.

Although both of these subsystems are of equal importance, this report is addressed to the capabilities and techniques concerned with the dynamics and the orthographic projection of bodies in motion.

Spacecraft

VISTA was originally developed as an aid in the checkout of an attitude computation program for the Orbiting Geophysical Observatory (OGO). The OGO spacecraft, depicted in Figure 1, carries into orbit about the earth, a large number of varied geophysical experiments. A greater understanding of the earth, and of earth-sun relationships will be obtained from them. The experiments will supply data on such phenomena as auroras and low energy solar particles.[3]

OGO has three main physical components: A Main Body, a Solar Oriented Experiment Package (SOEP), and an Orbital Plane Experiment Package (OPEP). In orbit the Observatory has five degrees of freedom: rotation of the Main Body about each of its principal axes, rotation of the SOEPs with respect to the Main Body, and rotation of the OPEPs with respect to the Main Body. The rotations of the Main Body about the

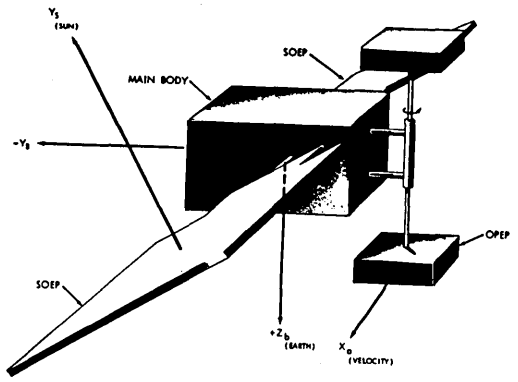


Figure 1—OGO spacecraft

longitudinal (Y) axis and about the solar array (X) axis are controlled so that the Body (Z) axis is directed toward the center of the earth. The rotation of the Main Body about the Z axis and the rotation of the solar array about its shaft axis are controlled so that the array face is aligned perpendicular to the sun line. The rotation of the OPEP with respect to its shaft is designed so that it tracks the component of the velocity in the orbital plane. Therefore, in order to define the orientation of the spacecraft there exist nine vectors, each with three components in inertial coordinates. Add to these 27 numbers other quantities of position, velocity, latitude, longitude, height, etc., and it can readily be seen that for any instant in time there is a large amount of interrelated data which is difficult, at best, to simultaneously correlate for analysis while at the same time verifying the overall spacecraft system response.

Coordinate systems

Within VISTA three coordinates systems, central or reference, body, and image, are used to describe the rotation and translation of the various bodies introduced into the viewing system and orient them in a proper viewing perspective.

Position and orientation information relating the body to the central coordinate system and similar criteria relating the image coordinate system to the reference system are sufficient to prepare the frames as shown in Figure 2. In the development of these frames (and the resulting motion picture) the central coordinate system was defined by information supplied from time sequenced orbit and attitude data on magnetic tape.

Such data as vehicle position, velocity and sun position are expressed as vectors in the geocentric equatorial inertial system. Spacecraft axes orientations, also expressed in this system, are given as unit vectors and

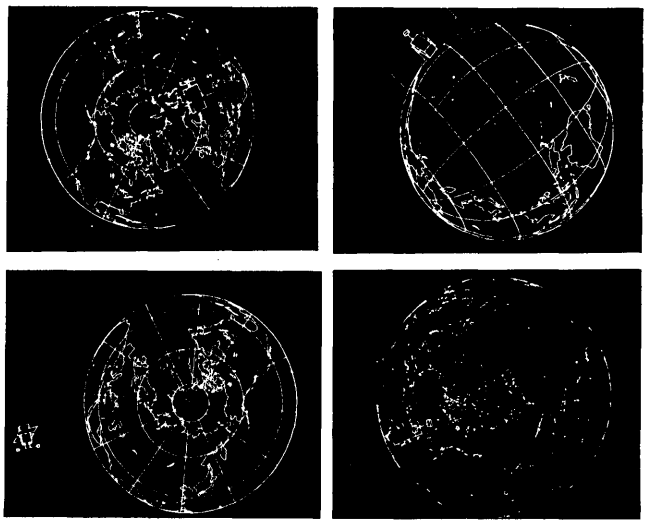


Figure 2—Projected frames

as such are directly applicable for use in the VISTA rotational matrices. Other information relating to the earth such as latitude and longitude of the subsatellite point are given in geodetic coordinates.

In VISTA, the image system is formed with the earth at the center and the viewer located on the + Z axis. The X, Y plane thus becomes the image raster "screen" to which all orthogonal projections are made. This image system may be rotated from the central inertial system by ordered angular rotations about one or more specified axes. Any inertial vector available in the data may be selected as the base view vector, i.e., vector along the line of sight of the VISTA "camera" taking the motion picture, and the appropriate angular rotations will be automatically generated. Additional rotations from the selected vector may be applied providing the capability of viewing the central system from any desired vantage point. In Figure 2 four views of the earth and the OGO spacecraft are shown for the same portion of the orbit.

Models

The geometry or shape of a non-spherical body is entered as a set of X, Y, Z points where each point describes the intersection of two or more lines or the intersection of a line and a surface. In order to reduce the complexity for intricate bodies, these points are numbered and plottable lines and body surfaces (polygon loops) are specified in terms of these numbers. When a body composed of hinged parts (e.g., the solar arrays on the OGO), requires that these parts vary with respect to each other then the points representing the "components" are grouped together. Several bodies may be introduced to the system in terms of points, lines, loops and components and projected simultaneously. Points grouped as components are rotated and translated to the image coordinate system where the line descriptors define plottable line segments and the loop descriptors are used in the final occultation testing.

The body represented in Figure 3 is a partial representation of the OGO spacecraft consisting of the main body and one solar array. The figure as shown is described with 14 points. Since the solar array rotates about the main body axis as it maintains alignment with the sun, points 1 through 8 are grouped as one component (the main body) and points 9 through 14 as another (the solar array). Seventeen lines and seven loops complete the description. As an example, the Y face of the main body is composed of lines (1,2), (2,3), (3,4), (4,1) and forms one loop.

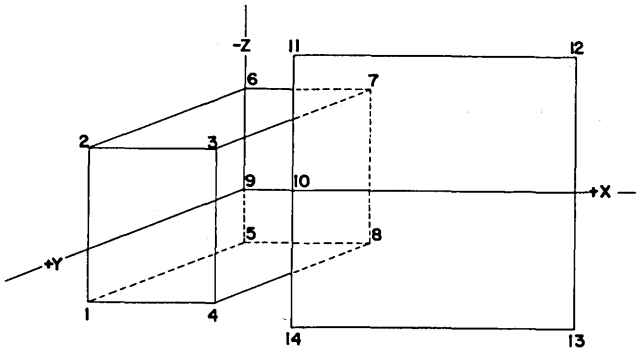


Figure 3—Partial OGO

This type of body description was a good balance between the ease of introduction of bodies into the computer (e.g., from scaled engineering drawings) and the ease of operation and projection in an internal computing sense. An interesting feature of this method is the ability to pass vectors (arrows) through the vehicle representative of such quantities as the magnetic field through which the vehicle is passing. Further, since the spacecraft is originally in alignment with the inertial axes, the rotation matrices to orient the spacecraft properly can be directly applied.

The earth as a spherical body is located at the center of the inertial coordinate system. A mapping of the shoreline features of the earth's surface is an additional input to the system consisting of approximately 8000 vectors. Optionally calculated and added to this mapping set are approximately 2500 vectors defining lines of longitude and latitude. (Samples of this representation of the earth are shown in Figure 2). The rotation of the earth in the VISTA model is driven by time as taken from the orbit data. The relative size and shape of the earth-spacecraft system is in part determined by the spacecraft position vector (magnitude) and in part by the location of view of the entire system.

In cases where it is desirable to include the moon or other major planets in the view, as with lunar orbits or approaches, these bodies are introduced into the system as spheres with appropriate surface mapping features. Proper position and orientation for these "special" bodies is achieved by utilizing ephemeris information and interpolating with the spacecraft time.

Hidden lines

As previously mentioned, one problem which had to be solved for the general case in order to achieve the desired effect was that of the hidden lines or occultation. This is a visibility determination; therefore, the selection of valid plottable line segments in the image coordinate system is performed last. All shoreline data as well as meridians, etc., are tested and those with negative Z values relative to the sphere center are immediately rejected. In this sense, spheres on which large amounts of data lie are handled in a somewhat different manner than the general manner applied to rectilinear figures. In the latter case, all lines associated with these figures that intersect others are broken into separate segments such that no two line segments intersect on the image plane. Examining the relationships of this new set of line segments and the body surface loops is sufficient for determining valid lines. A line segment is invalid if its midpoint falls behind and inside some loop; otherwise, it will be plotted.

This test for line validity is accomplished in two separate steps. The first is the "inside," "outside" determination; the second, the "front" or "back." Two different algorithms have been used for the first test—both successfully. Both solutions rely on lines being constructed on the image plane from the point to be tested (line midpoint) to each vertex of the particular polygon loop forming a set of triangles. One method computes and sums the interior angles about the test point. If the sum is equal to 360°, the point lies inside the polygon and may not be plotted. The other method computes in order the triangular areas. A change in sign in the area of any triangle determines that the point lies outside the polygon and will be plotted.

The second and final test for line validity is accomplished by constructing a line parallel to the view vector from the test point to the loop plane. If the point of intersection has a larger Z value than the test point itself, the point lies behind the plane and will not be plotted. Several optimization techniques are employed which effectively reduce needless validity testing. One such technique first examines minimum and maximum component or body values projected on the image plane and avoids body to body occultation testing when appropriate.

Dynamic control

All system parameters can be modified from the computer operator console as the system is running. In addition, selected parameters (view, zoom) may be assigned to continuous alteration on a frame by frame basis and controlled by the computer sense switches. Thus additional bodies may be introduced to the system and current ones modified. In this dynamic fashion, spacecraft and booster separation may be performed with relative ease. All transitions between views are made smoothly so that no discontinuity is noticed when the film is being projected.

Future applications

VISTA or VISTA-like computer techniques have several obvious applications, and, with some imagination, several not so obvious extensions. Two notable applications in space research are for manned spaceflights and for a galactic probe.

For manned spaceflight, VISTA, modified for real-time processing, would allow ground based observers to "see" docking maneuvers, rendezvous, or even a lunar touchdown. For these applications the user's facility for controlling the field of view would enable the viewer to observe the surrounding environment from any desired vantage point. The value of a viewing capability in rapid detection and analysis of anomalies in spacecraft motion in immeasurable.

Another application of VISTA would be the display of the orbit of a galactic probe. To show such an orbit the coordinate system used could be heliocentric (sun centered) rather than geocentric (earth centered). The system view chosen could be approximately normal to the planetary orbital planes. In viewing a motion picture of such an orbit not only would a time compression occur (orbital period measured in years, viewing time in minutes) but the relative locations of planets to the spacecraft and their influence could readily be seen and appreciated.

A more down to earth application of an extended VISTA is in air traffic control. The motion depicted on a CRT and the change of reference capabilities could significantly augment the controller-radar system. As an example the center of the coordinate system (radar) for a controller would normally be the tower. However, if any doubt arose concerning the vectoring of two aircraft the view of the controller could immediately be changed for better definition of the situation. If the same radar information could be fed to hybrid computers aboard each aircraft and viewed there, the cen-

ter of the coordinate system could be the individual aircraft and the pilot could "see" what was in his vicinity as well as the directions of travel. Possible, yes—practical is another question.

SUMMARY

The VISTA system is designed to aid the space science analysis effort by preparing motion pictures which are realistic representations of the desired situation. This type of visual presentation permits a rapid correlation and intuitive understanding of the relationships of dynamic bodies and subjects all data to a more rapid and comprehensive analysis in a manner not available through reading lists of numbers or even plots of some particular parameter.

ACKNOWLEDGMENTS

Many thanks are expressed to Michael Mahoney of Goddard Space Flight Center for his original concepts and to Joel Erdwinn of Computer Sciences Corporation for his "hidden line algorithm" suggestions.

REFERENCES

1 F H HARLOW J P SHANNON J E WELCH
Science 149 1092 1965
B. R. GROVES *Science* 155 1662 1967
K C KNOWLTON *Science* 150 1116 1965
R A WEISS *Journal of the ACM* 13 194 1966
E E ZAJAC *Journal of the ACM* 7 169 1964
Edited by F F KUO J F KAISER
System analysis by digital computer
John Wiley and Sons, Inc chap 11 p 375 New York New York 1966
R A SIDERS
Computer graphics
American Management Association chap 11 p 148 New York 1966

2 M MAHONEY J QUANN
Visual presentation of the motion and orientation of an orbiting spacecraft (OGO)
NASA TN D-2918 1965

3 G H LUDWIG
The orbiting geophysical observations
NASA TN D-2646 1965

Current status of large scale integration technology

by RICHARD L. PETRITZ
Texas Instruments Incorporated
Dallas, Texas

I. INTRODUCTION

Considerable progress has been made in large scale integration technology during the past year. Many of the goals, which were theoretical assumptions last year, are now well along the way to reality. Accomplishments range from basic materials processing improvements to systems architecture innovations. While this paper will concentrate on large scale integration technology achievements, we shall also focus some attention on progress in significant related areas.

The electronics industry has been relatively open with respect to large scale integration (LSI) investigations, and much of the work has been, and continues to be, presented at technical meetings and published in the literature. In addition, the U.S. Air Force has under sponsorship a major program[1] with three contractors[2,3,4] to develop LSI technology. We shall make frequent reference to these reports and to the author's papers of 1965[5] and 1966.[6]

Reviewing the terminology which the author used in previous papers,[5,6] we see LSI as a system of technologies underlying the products which are called IECs (integrated electronic components).* The distinction between "technology" and "product" is summarized in Table I for three generations of solid-state electronics. Table I shows "transistor" as the surviving product terminology of the second generation of electronics technology, and lists some of the technologies. Similarly, "integrated circuits" is the surviving product terminology of the third generation. As discussed in Ref. 6 a distinction should be made between "integrated circuits" (IC) and the products, namely, "integrated electronic components," (ICE) that will result from large scale integration technology. The principal distinction between an IC and IEC is that the latter is the result of interconnecting circuits within the structure, whereas an IC is the result of interconnecting devices within the structure. Figure 1 shows the distinction between device, integrated circuit, and integrated electronic component.

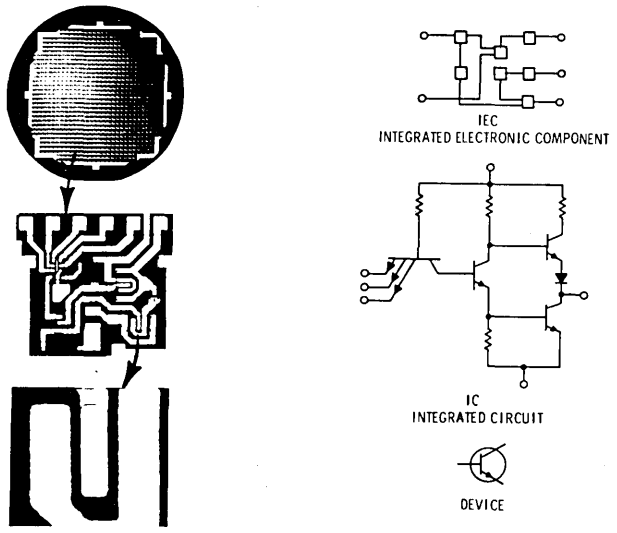


Figure 1 – Pictorial view of integrated electronic component (IEC), integrated circuit (IC), and semiconductor device

Another term that is being used to some extent is MSI (medium scale integration) along with LSI. When used in the context of complexity level MSI is generally related to complexity levels of 10-100 circuits, while LSI refers to complexity levels greater than 100 circuits.

At least three well-defined LSI technologies are under development:

LSI chip technology
LSI hybrid technology
LSI full-slice technology

*In reference 6, IEC was interpreted to mean "integrated equipment component." Because of the broad acceptance of the term "integrated electronics" as the generic term of the industry in place of "microelectronics," we now prefer IEC to mean "integrated electronic component." Another term under consideration is integrated electronic device (IED); however, we shall use IEC in this paper.

Table I. Technology — Products

Generation of Electronics	Technology Terminology	Product Terminology
2nd	grown junction alloy mesa planar bipolar MOS	Transistor
3rd	monolithic hybrid thin film thick film	Integrated Circuit (IC)
4th	LSI technology chip— • 100% yield over chip area • fixed pattern metalization • customized wiring • single or few chips per package full slice— • fixed pattern metalization • discretionary wiring for yield enhancement • customized wiring • redundancy hybrid— • interconnection of chips through use of film technologies • customized wiring	Integrated Electronic Component (IEC)

The basic characteristics of these technologies are summarized in Table I, and Figure 2 illustrates them.

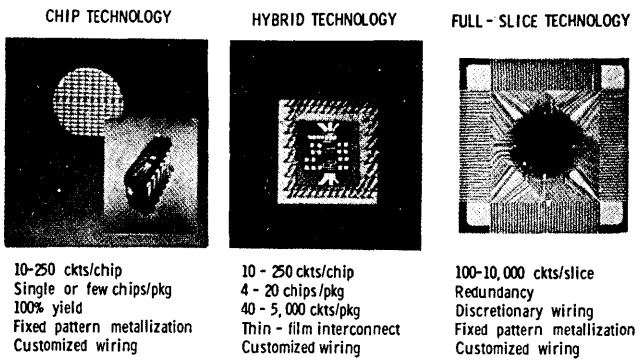


Figure 2 — Pictorial view of large scale integration technologies. The pictures represent comparable sizes

Because of confusion between the terms fixed pattern metalization, discretionary wiring and customized wiring, and the need for preciseness in their meaning, we will review these and related concepts. First let us emphasize that we are discussing two classes of products, standard and custom. The term customized wiring (metalization) refers to a customized metalization pattern for a specific product.

Looking to the manufacturing process, we need to recognize two major technologies, fixed pattern metalization, and discretionary wiring (pattern) metalization. The fixed pattern process uses the same metalization pattern from slice to slice in the manufacture of a specific product, whether it be a standard or a custom product. For a custom product, the fixed pattern metalization differs from product to product, but for the manufacture of a specific product it remains the same.

Discretionary-wiring technology is a method of enhancing yield and provides different metalization patterns for each slice in the manufacturing process, whether the product be standard or custom. A particular advantage or the discretionary-wiring technology is that for many applications it accomplishes the customized-wiring function with relative simplicity. However, in other applications this is not necessarily the case.

In summary, both standard and custom products can be manufactured with a fixed-pattern technology or with a discretionary-wiring technology, or a combination of both. In order to abbreviate terminology, we have defined on Table I chip technology to imply fixed pattern metalization, and full-slice technology to employ both fixed pattern metalization and discretionary wiring. As will be discussed later, the relative emphasis of fixed versus discretionary metalization varies for different full-slice technology IECs.

Progress will be reviewed in each of these technologies as follows:

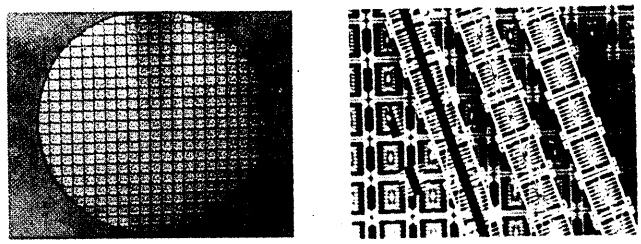
II LSI Bipolar Chip Technology
III LSI Full-Slice Technology
IV LSI MOS Technology
V LSI Hybrid Technology

Section VI develops the considerations of complexity (level of integration) versus cost, performance, and reliability for LSI technologies.

II. LSI bipolar chip technology

A. Review of Status of 1967 IC and IEC Production Technology

The monolithic integrated circuit technology processes slices of silicon such as that shown in Figure 3a. The slice has been processed through metalization and is ready for probing. Each small area (chip or bar) on the wafer is probed, and the good and bad units are marked accordingly. The slice is then scribed into chips and the good chips in turn are assembled into packages and tested. Figure 3b shows a sequence of the silicon chip being assembled into a plastic package.



(a) (b)

Figure 3—a) Semiconductor slice showing metalized integrated circuit chip areas. The slice is ready for probing and then scribing into chips. (b) From left to right, fourteen lead metal frame without integrated circuit chip; the chip mounted in the lead frame; the plastic encapsulation has occurred; the completed integrated circuit in a plastic package

Table II is a listing of some of the key characteristics of the Series 54 T²L line of integrated circuits. The important points are the following: bar (chip) sizes average 2600 mil²; the devices per bar average 25; the area per device averages 104 mil²; the average number of circuits per package is 4; and the average area per circuit is 620 mil². These averages include the effect of bonding pads. Line widths of 0.2 mil and spacings of 0.2 mil are used.

These data summarize modern slice processing technology. Integrated circuits of 1962-63 vintage would show one or two circuits per bar and the bars would be considerably larger. For example the early Series 51 designs used bar sizes 14,000 mil² (70 mil × 200 mil) and contained one to two circuits. Line widths of 1 mil and spacings of 1 mil were used.

The industry recognized at least three years ago that four circuits were about all that could be economically placed in a single package if all gate (circuit) leads needed to be accessible from the terminals. Four circuits require 16 terminals, using 4 terminals per circuit. At the same time it was also reasonably clear that greater economics were ahead if more circuit function could be incorporated into a single package. Thus the direction was taken toward interconnecting circuits within the chip, and this was the genesis of the integrated electronic component product line.

Table III lists the Series 54 T²L IECs that are now in production. Note that the average bar size is 7000 mil²; the average devices per bar is 124; the average area per device is 56 mil²; the average circuits per bar is 25; and the average area per circuit function is 280 mil². By comparing these averages with those of Table II it is apparent that IECs are more effective than ICs in utilizing the silicon area. Note the trend to larger bar sizes, with several approaching 10,000 mil².

Table II. Series 54/74 T^2L Integrated Circuits (ICs)

Device No.	Circuit Type	Bar Size (mil²)	Bar Size (mil²)	No. of Devices	Area/Device (mil²)	No. of Circuits	Area/Circuit (mil²)
SN5400	Quad — 2 input gate	50 x 60	3000	36	83	4	750
SN5410	Triple — 3 input gate	50 x 60	3000	27	110	3	1000
SN5420	Dual — 4 input gate	45 x 45	2025	18	125	2	1012
SN5430	Single — 8 input gate	40 x 40	1600	9	180	1	1600
SN5440	Power dual — 4 input gate	50 x 50	2500	22	114	2	1250
SN5450	Dual 2-wide 2 input AND-OR-Inv. E	50 x 55	2750	24	115	6	460
SN5451	Dual 2-wide 2 input AND-OR-Inv.	50 x 55	2750	24	115	6	460
SN5453	4-wide 2 input AND-OR-Inv. E	50 x 55	2750	18	153	5	550
SN5454	4-wide 2 input AND-OR-Inv.	50 x 55	2750	18	153	5	550
SN5460	Dual — 4 input Expander	35 x 40	1400	6	230	2	700
SN5470	J-K Flip-Flop	55 x 60	3300	56	57	8	400
SN5472	J-K Flip-Flop Master Slave	55 x 60	3300	40	80	6	535
	TOTAL		31,125	298	—	50	—
	Average		2600	25	104	4.2	620

Table III. Series 54/74 T^2L Integrated Electronic Components (IECs)

Device	Function	Bar Size (mil^2)	Bar Area (mil^2)	No. of Devices	Area/Device (mil^2)	Equivalent Gate	Area/Gate (mil^2)
SN5441	BCD to Decimal Decoder/Driver	60 x 60	3600	50	72	17	212
SN5475	Quad latch	60 x 120	7200	120	60	24	300
SN5480	Gated full adder	65 x 65	4225	69	61	14	301
SN5482	2-Bit full adder	65 x 65	4225	83	51	21	200
SN5490	BCD decade counter	50 x 115	5750	102	56	18	320
SN5491	8-Bit shift register	55 x 110	6050	143	42	35	173
SN5492	Divide by 12 counter	50 x 115	5750	96	60	17	340
SN5493	Divide by 16 counter	50 x 115	5750	96	60	17	340
SN5494	Dual P.I., S.O. 4-Bit S.R.	70 x 110	7700	125	62	20	385
SN5496	P.I., S.I., P.O. 5-Bit S.R.	70 x 140	9800	158	62	24	450
SN1286	P.L. Serial 5-Bit ring counter	70 x 140	9800	169	58	30	327
SN1287	Dual P.L. Count to Zero 5-Bit R.C.	70 x 110	7700	153	50	33	230
SN1288	Dual P.L. S.S., 5-Bit R.C.	70 x 140	9800	191	51	30	327
SN5484	Active Element Memory (AEM)	60 x 120	7200	100	72	40	180
SN5495	4-Bit UP/DOWN Shift Reg.	70 x 120	8400	156	54	33	255
SN5497	Synchronous BCD decade counter	75 x 120	9000	171	53	28	320
	TOTAL		111,950	1982	—	401	—
	Average		7000	124	56	25	280

Considerable improvement is attained in the efficiency of the circuit technology for IECs compared to ICs. One aspect of this can be understood with reference to Figure 4. The basic Series 54 IC gate is shown at the top left of the figure, while variations of this gate used in the Series 54 IECs are shown in the other three parts of the figure. For example, the internal/expander gate allows for wired-ORing on the chip. Also, since it is used only on the chip, it does not need the high driving capability required when coming off the chip, and requires only 3 devices compared with 9 for the basic Series 54 gate. For the latter, the totem-pole output is used in order to drive capacitive loads. The IC gate must be designed with many of these factors in mind and of necessity involves compromises.

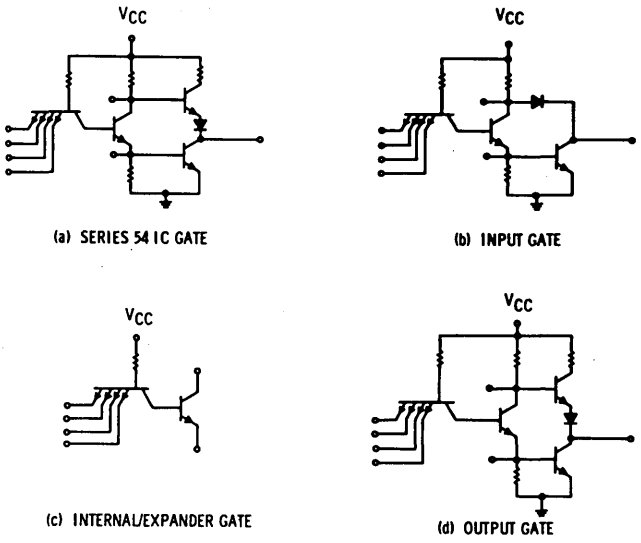


Figure 4 — Circuits used in Series 54 T²L ICs and IECs

We can summarize at this point by stating that the semiconductor industry has moved to higher levels of integration by the combined effect of employing high resolution technology so that less area is required for a device, and by utilizing chips of larger area.

To what degree can we expect to increase the complexity levels attainable on chips, while at the same time gain in over-all savings? This question has been studied at Texas Instruments during the past year. Similar studies have been reported by Fairchild[7] and RCA[3] and no doubt are being carried on throughout the industry. Because of the proprietary nature of process-yield information, we are not at liberty to reveal the detailed information of the study; however, we can report certain information that will be of interest.

B. Yield versus area

The study was conducted using Series 54 production data. Two particular goals were emphasized: (1) the establishment of the dependence of yield on chip area, and (2) the establishment of the major yield loss mechanisms. Item 1 would allow for projection of chip areas into the future, based on today's technology, while item 2 provides a basis for improving over-all yield. This latter subject goes beyond the scope of this paper, so will not be discussed further.

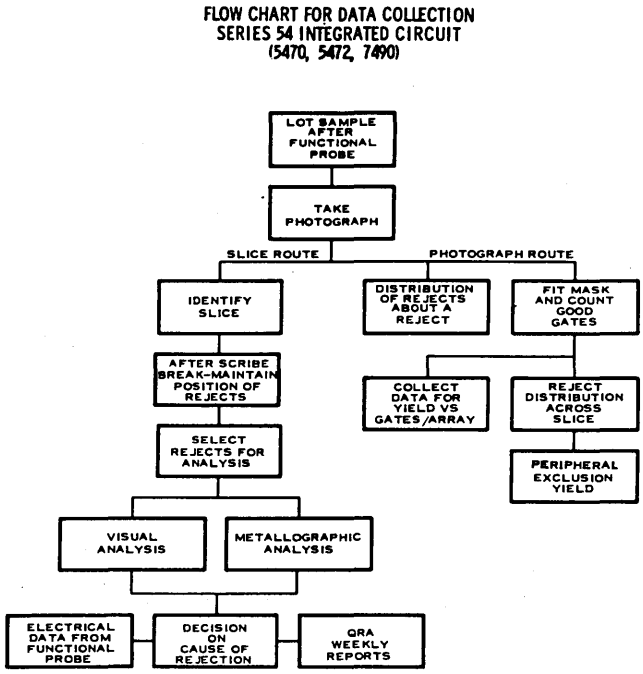


Figure 5 — Flow chart defining yield versus area study (photograph route) and yield loss mechanism study (slice route)

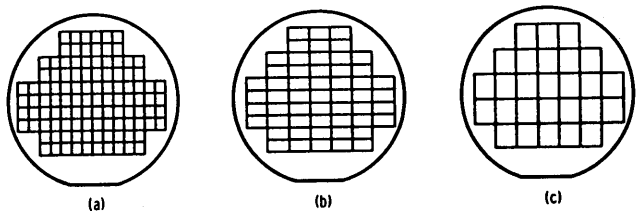


Figure 6 — Masks defining areas for yield versus area study

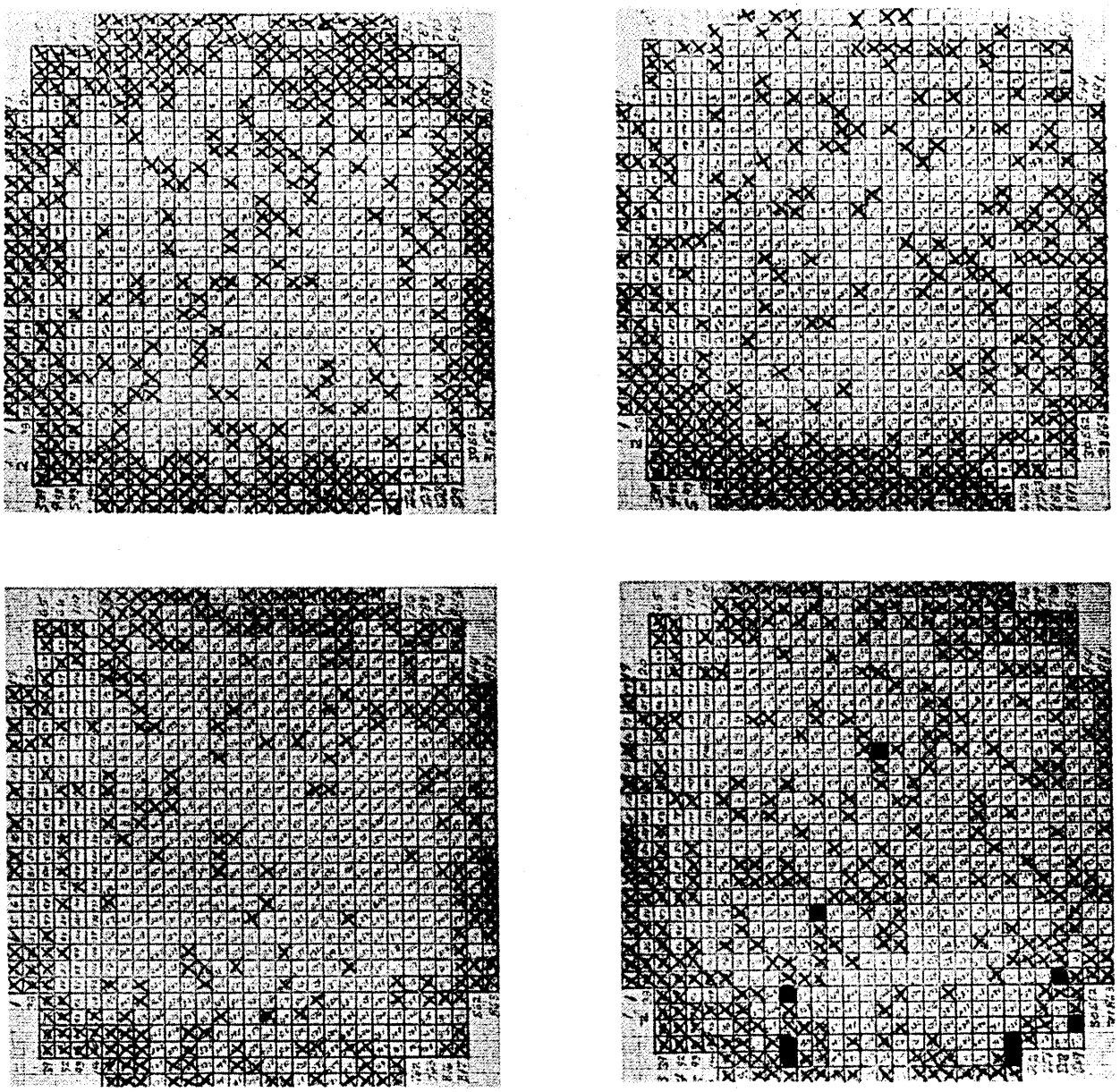


Figure 7—Typical slices in yield versus study—the X's designate faulty units

Figure 5 diagrams the flow chart for data collection and analyses. Photographs of slices after functional probe were studied to determine yield as a function of area. The three products studies were two flip-flops of area 3300 mil² (5470 and 5472) and one IEC of area 5750 mil² (5490). In order to study the effect of area on yield, a set of overlay masks was defined as shown in Figure 6. Note that each mask defines an area twice as large as the previous one. By counting the good units within the area defined by the overlay mask and dividing by the total number of units on the slice, the yield as a function of bar area is attained. Actual slices are shown in Figure 7.

The data of yield versus area were plotted on semi-log paper. The simple theory of random defects predicts a dependency:

$$Y = B \exp(-\lambda \text{ Area}) \qquad Y = \text{yield}$$
$$\text{Log } Y = \text{Log } B - \lambda \text{ Area} \qquad \lambda = \text{average defect density}$$
$$B = \text{non-random yield loss}$$

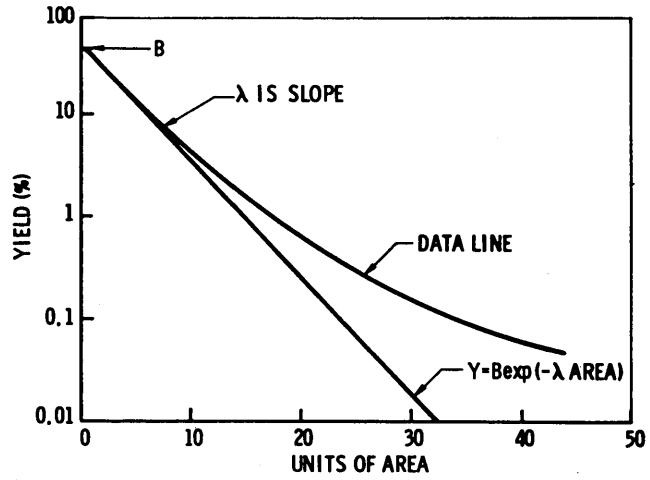


Figure 8 – Plot of yield versus area (relative)

This defines a straight line function on semi-log paper as shown in Figure 8. The actual data followed a line of the general shape shown by the data line of the figure. Because of proprietary considerations, the abscissa of Figure 8 is relative – not actual area. The important conclusion established by this study is that yield holds up for larger areas than simple theory predicts. In a qualitative sense we interpret that good units tend to cluster, and, likewise, defects tend to cluster, Examples in Figure 7 show this quite graphically; there are relatively large areas free of defects, and areas where defects cluster. This is particularly so around the slice periphery.

This study of yield versus area has led us to the conclusion that the LSI chip technology has by no means exhausted itself at the areas of 10,000 mil² represented in Table III. Using the data obtained in this study, along with some reasonable forecasting of yield improvement that will take place during the next few years, we forecast that chip technology will be useful for areas as large as ¼ in. × ¼ in. = 62,500 mil².

The other major factor that governs complexity level on a chip is the area required for a device. We have seen over the past five years significant improvements in optical technology and we forecast that improvements will continue to be made. For further discussion of optical factors the reader is referred to the discussion of Figure 21 in Ref. 6.

The combined effect of improved yield such that larger chip areas will be useful, along with the smaller areas required for circuits, leads us to forecast the following T²L complexity levels for the 1970's:

Logic: 250 gates/chip

Memory: 500 bits/chip

Texas Instruments has a program to reach these complexity levels by an extension of the Series 54 technology to chip sizes of 62,500 mil², retaining today's device geometries. Thus we have:

Chip size – 62,500 mil²

Logic – 250 mil²/gate→250 gates/chip

Memory – 125 mil²/gate→500 bits/chip

New designs which are geared for production in the 1970's should consider these complexity levels as attainable goals. The actual way the complexity level is achieved may differ somewhat from the above figures, e.g., smaller device geometries could be used resulting in smaller chips.

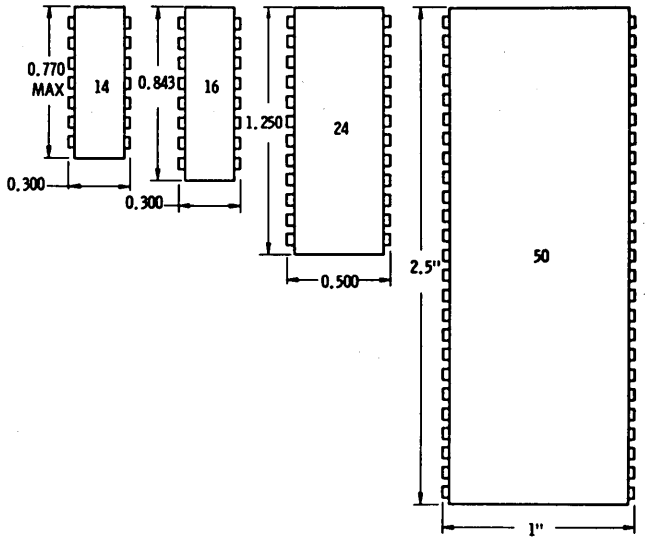


Figure 9 – Family of plug-in IC and IEC packages

In summary, yield has improved to where, in 1967, IECs of 10,000 mil² containing 35-40 T²L logic circuits are now in or near production. Designs aimed for production in the seventies should comprehend chips containing up to 250 logic gates or 500 memory bits. Packages are under development as shown in Figure 9 with up to 50 pins per package to accommodate these chips.

C. Custom products versus standard products

The question of custom products versus standard products is not a new one to the semiconductor industry. At the device level many custom transistors are manufactured exclusively for a single customer. Integrated circuits in their early phases attempted to solve the custom-product problem by a master slice with mixtures of transistors, diodes, and resistors which could be interconnected by specific metalization patterns to provide a specific circuit for a customer. While the master slice has proved effective for varying a basic circuit, it has not been used for a broad range of circuits. Instead, the problem of cus-

tom versus standard products has been handled differently. Custom lines have been designed, developed, and placed into production for large customers whose total business warranted the expense of this approach. Often these custom developments have led to standard product lines patterned after them, and a large part of the industry requirements has been satisfied by standard product lines.

However, we must not conclude that a similar course will necessarily follow for IECs. A useful figure for analysis of the question was recently published by IBM.[8] Figure 10 plots the number of unique parts versus level of complexity (level of integration) for Central Processing Units (CPUs) of 1 K, 10 K, and 100 K circuits, respectively. We note that for complexity levels of 3-4 circuits the number of unique parts is relatively small. This is a key reason for the wide acceptance of standard product integrated circuits.

Complexity levels from 10-250 circuits pose the most difficult part-number problem, since high numbers of unique parts imply relatively small usage of the corresponding parts.

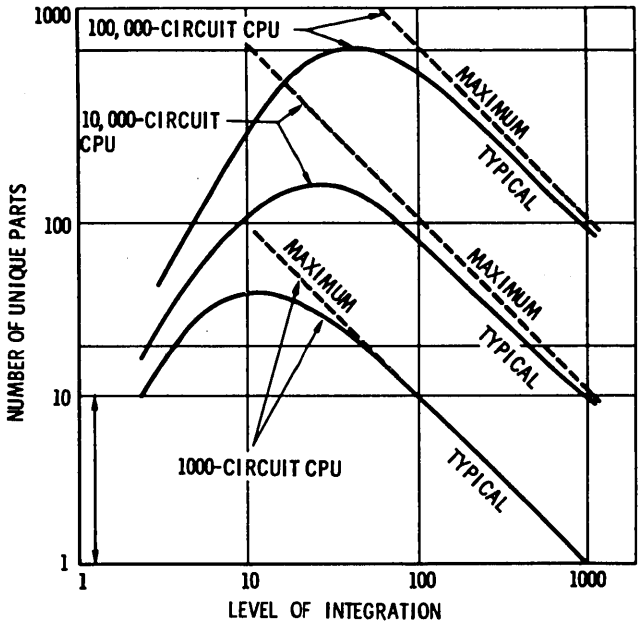


Figure 10—Plot of number of unique parts versus level of integration

Thus we conclude that an efficient, low-cost, quick-turnaround method will be needed to supply custom IECs in the complexity range of 10-250 circuits. There are many different approaches—but all appear to focus on three basic premises: the first is that a good computer-aided design capability be established so that low-cost masks can be designed in a short time. Secondly, automatic artwork generation and mask making are required. Finally, some form of master-slice technology is needed to lessen the processing expense and to shorten the turnaround time.

Fairchild has described a master-slice approach for bipolar (DTL) technology. Master chips of 8800 mil^2, containing 32 three-input NAND gates, have been defined. One can expect these chip areas to increase in size to allow for greater complexity levels.

RCA has chosen ECL as its basic approach for LSI logic technology. Under their Air Force contract[3] RCA has set goals of chip sizes to reach the ¼" x ¼" level. The gate areas will approach the 250 mil^2 size—thus complexity levels up to 250 logic gates per chip are the goal. Their progress reports[3] show working arrays of 23,000 mil^2 containing 125 transistors and 90 resistors.

Texas Instruments has defined a master-slice approach as a companion approach to the standard IC and IEC Series 54 chip technology. Table IV summarizes this program. Since this program is representative of the custom-wired LSI chip-technology programs which industry is developing, we shall detail some of its aspects.

Figure 11 shows an area 240 × 240 mil (57,600 mil^2) providing 256 gates, for an average area/gate of 225 mil^2. Note that a 60 × 60 mil^2 unit area is stepped across the slice. Master slice No. 1 (Figure 12) has 16 gates in the 60 × 60 mil^2 area. There are 4 input gates, 8 internal/expander gates, and 4 output gates. The devices are interconnected to form these circuits with the first level of metalization as shown in Figure 12a; pads are shown for interconnecting these circuits into specific IECs. The mask for feed-through holes is shown in Figure 12b, the second-level metal in Figure 12c, and the resulting function in Figure 12d. The design rules for this program are summarized in Table IV.

The three approaches outlined above, Fairchild-DTL, RCA-ECL, and TI-T^{2}L, while varying in detail, have all taken the same basic approach; namely, a master slice which has all diffusions made as the standard item. Personality or customization is imparted through first- and second-level metalization.

The key technological advances required for a successful customized LSI chip technology include: improvement of yield such that chip areas 30-60 K mil^2 can be economically used, two-level metalization technology is required for crossover capability, and finally computer-aided design capability must be developed so that a customer's logic equations can be translated directly into a mask layout for the interconnection wiring.

Table IV. Custom Series 54/74 IEC Program

I. Purpose:

1. To supply custom digital logic arrays in the 15 to 250 gate complexity range for low cost, low volume requirements.
2. 4-week cycle time from receipt of customer order to shipment of prototypes.

II. Approach:

1. Master slice with fixed second-level lead patterns interconnecting from 2 to 16 bars.
2. Series 54/74 circuits.
3. 14, 16, 24 and 50 pin dual-in-line headers.

III. Schedule:

	Weeks
1. Array logic design and mask design	1
2. Photomasks from photolab	1
3. Material processing and assembly	1
4. Prototype testing and evaluation	1
Total	4

IV. Design Rules:

Package	Bars		Gates		Gates/Pin	
	Min.	Max.	Min.	Max.	Min.	Max.
14	1	3	15	45	1.07	3.2
16	1	3	15	45	0.94	2.8
24	2	8	30	120	1.25	5.0
50	8	16	120	240	2.40	4.8

III. LSI full-slice technology

We at Texas Instruments feel that the full potential of semiconductor technology for integrated electronics will be realized only when the entire semiconductor slice constitutes the packaged product. Arguments for this position include: (1) the full slice is the natural working unit of semiconductor technology, and (2) very high complexity levels (1-5 K circuits) are available directly on the slice of silicon.

To this end we have a program, in part under Air Force sponsorship,[2] to develop LSI full-slice technology for IECs. Recognizing the limitations of yield, this program has sought methods for producing working electronic functions which did not require 100% yield over the full silicon slice; two main avenues have been taken:

(1) Discretionary wiring
(2) Redundancy

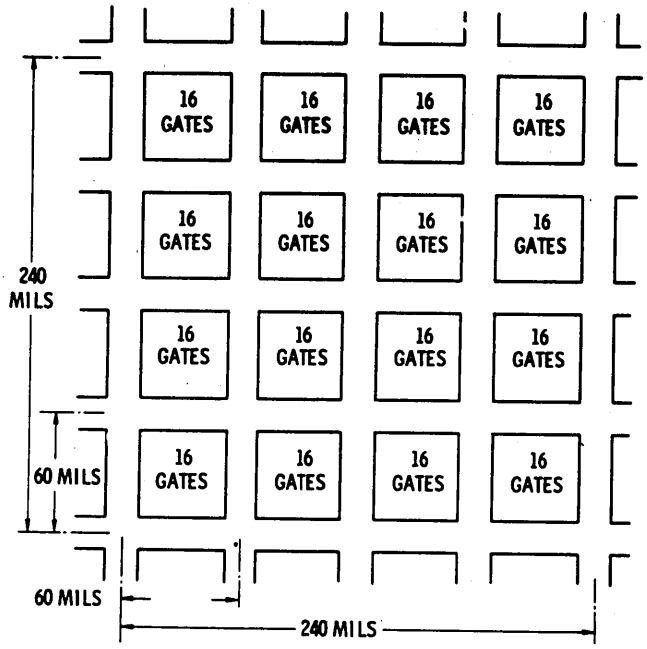


Figure 11—Chip areas defined for custom Series 54 IEC program

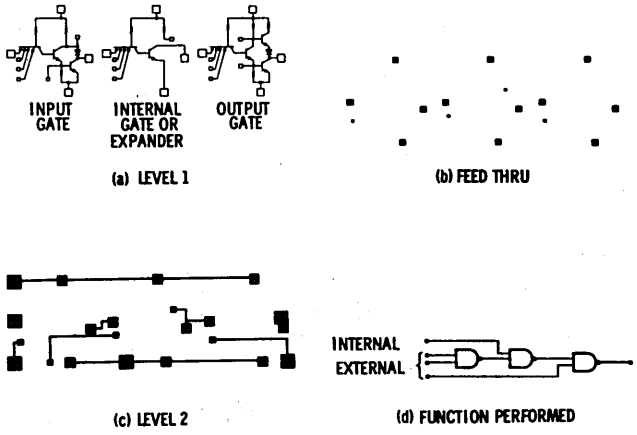


Figure 12—(a) Basic circuits for custom Series 54 IEC program; (b) feed-through mask; (c) second-level metalization mask; (d) function to be performed

The concept of discretionary wiring is one of probing to identify the good or useful circuit elements on the slice and then interconnecting them to make the final function. Redundancy has been incorporated into the design philosophy of the memory program so that discretionary wiring has been reduced to that of a single mask per slice.

The Air Force-Texas Instruments LSI program calls for the development and construction of a research vehicle, which is a computer for a terrain-following radar system. This computer consists of four main parts: a general-purpose computer, an active memory (Read-Write and Read-Only), a phase shift computer, and a video integrator. The characteristics of these four parts and their implementation in terms of LSI technology are summarized in Table V. More detailed characteristics of the logic arrays that have been defined at this writing are summarized in Table VI. Twelve logic functions are listed, with the number of gates per function, the number of flip-flops, and the number of equivalent gates (where one flip-flop is equivalent to four gates) per function. We see that the circuit complexity level varies from 134 to 262, with 193 the average. Ten of the twelve functions are used only once, the other two part-numbers having high usage. This illustrates the need for customized wiring at this level of circuit complexity and confirms the conclusion of Figure 8.

The program to develop the full-slice LSI technology includes work aimed at standard-product IECs and custom-product IECs. The Read-Write memory has served as the vehicle to develop discretionary-wiring technology for standard-product IECs with complexity levels greater than 1000 circuits/slice. The Read-Only memory has provided a vehicle in which both customized wiring and discretionary wiring are employed together in a relatively simple manner. The logic portion of the computer has served as the vehicle for developing the technology for customized and discretionary wiring of a more sophisticated nature than memory. We shall review both the memory and the logic programs.

A. Read-write memory; standard-product IECs

The Read-Write memory offers a high-volume application for standard-product IECs. It also affords the opportunity for incorporating complexity levels of the 1-5 K bits per package while maintaining a relatively small number of pins on a package. This is accomplished by doing address decoding on the slice. Figure 13 plots pin connections versus cell complexity for memory. The various lines show how incorporation of address decoding and/or other functions on the slice keeps the number of external pin connections to relatively small numbers; for example, 5000 bits of word-organized memory can be accessed in a 150-pin package.

The basic memory slice is shown in Figure 14. Note that $60 \times 64 = 3840$ potential storage bits are provided, with up to 60 word drivers and decoding gates. The basic circuitry is illustrated in Figure 15a, the storage cell being two cross-connected multi-emitter (T^2L) transistors and two load resistors. The use of redundancy allows for 13 of 16 bits in each

Table V. Research Vehicle—Computer for Terrain-Following Radar

Part	Characteristics
GENERAL PURPOSE COMPUTER	16-bit word length 2-MHz clock rate 34 logic arrays of 21 wiring configurations (b1-b21) of usage and complexity shown in Table VI. Total, 5519 gates, 400 flip-flops; average gates/pin = 2.4.
MEMORY	Read-Only, 512 words of 32 bits, non-volatile, stores program and essential parameters. Read-Write, 128 words of 32 bits, volatile, stores radar data being processed. System access time, 2 μ/sec. Slice Technology: 16 Read-Only arrays, 1024 bits/array 4 Read-Write arrays, 1024 bits/array 2 Sense amplifiers, bit driver arrays, 32 SA/array, 32 bit drivers/array 9 Logic arrays for decoding and address register, average complexity 200 gates/array 31 Arrays for memory
PHASE SHIFT COMPUTER	14 Type b22 arrays (Table VI) 1 Type b23 array (Table VI)
VIDEO INTEGRATOR	6 Logic arrays b24-b29 (Table VI) 5 Memory arrays

Table VI. Logic Partitioning for LSI Computer

Logic Function	No. of Gates	No. of Flip-Flops	Equiv. Gates, 1-FF = 4 Gates	Usage
b-01	138	10	178	16
b-02	148	20	228	1
b-03	139	27	247	1
b-04	158	26	262	1
b-22	80	28	192	14
b-23	162	14	218	1
b-24	124	22	212	1
b-25	66	26	170	1
b-26	49	24	145	1
b-27	62	27	170	1
b-28	38	24	134	1
b-29	68	24	164	1
Total	1232	272	2320	40
Average	103	23	193	

section of a column to be used. Thus actual word lengths up to 52 bits may be employed. The original design of the Air Force contract called for 32 words of 52 bits, or 1664 bits per slice. Because of a change in systems design the present slice utilizes 32 words of 32 bits, or 1024 bits per slice as noted in Table V. Figure 15b shows the layout of the storage bits, the resistors on the right, the transistors on the left; note that only 127 mil^2 is required per bit. Figure 15c is an expanded view of the slice before metalization; the decoder and word drivers are at the top and the storage bits are at the bottom. Figure 15d is an expanded view after first level metalization. Vertical word lines which interconnect 16 bits in a column are put down as part of the first level metalization. Redundancy is so employed that only 13 of the 16 bits are required. In the next step a thin insulating layer of SiO_2 is deposited and feed-throughs opened. Next, metal is evaporated or sputtered over the entire slice, and selectively removed so that feed-throughs are brought from the first level to the top of the slice. All masking operations through this point have been fixed patterns. No testing and no discretionary masks have been involved.

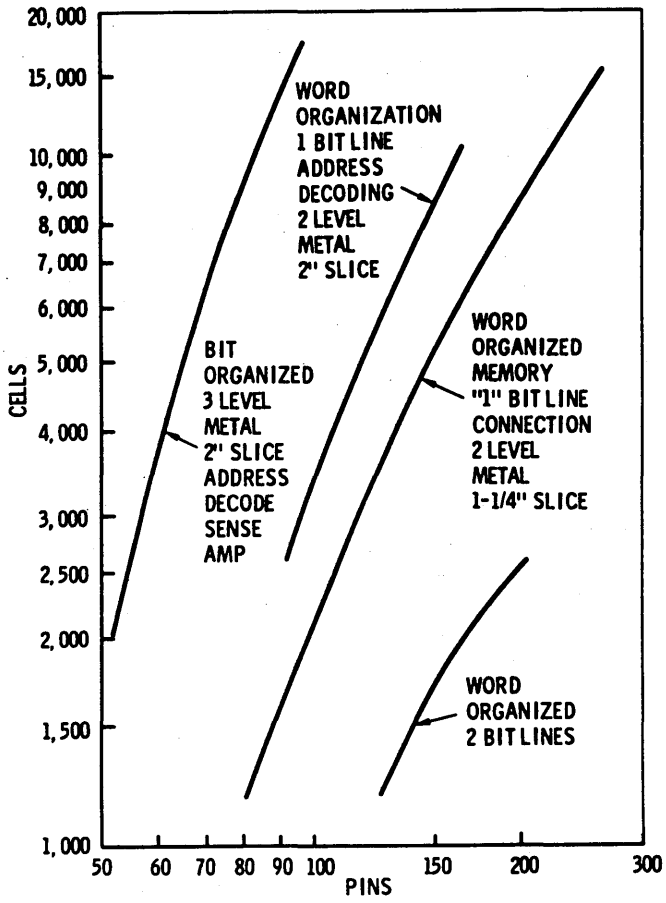


Figure 13—Plot of number of memory cells versus number of pins on package for different active memory organizations. The various curves show the reduction in number of pins required by incorporating address-decoding and other functions on slice

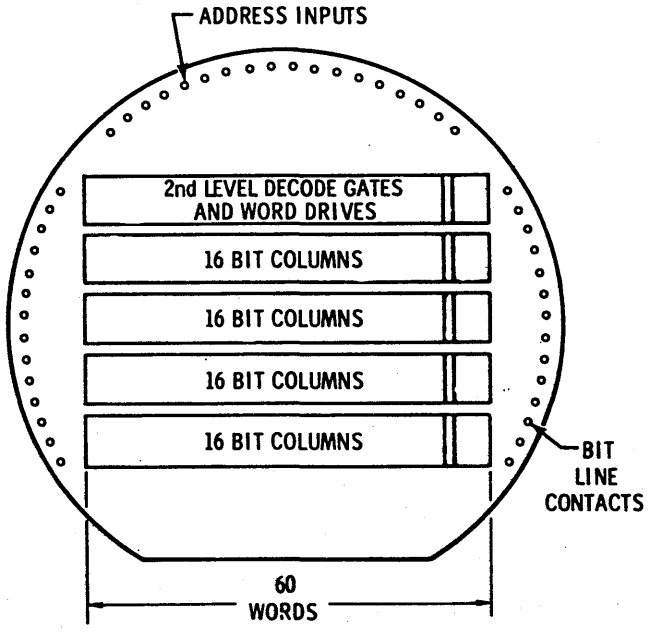


Figure 14—Organization and layout of semiconductor active memory slice.

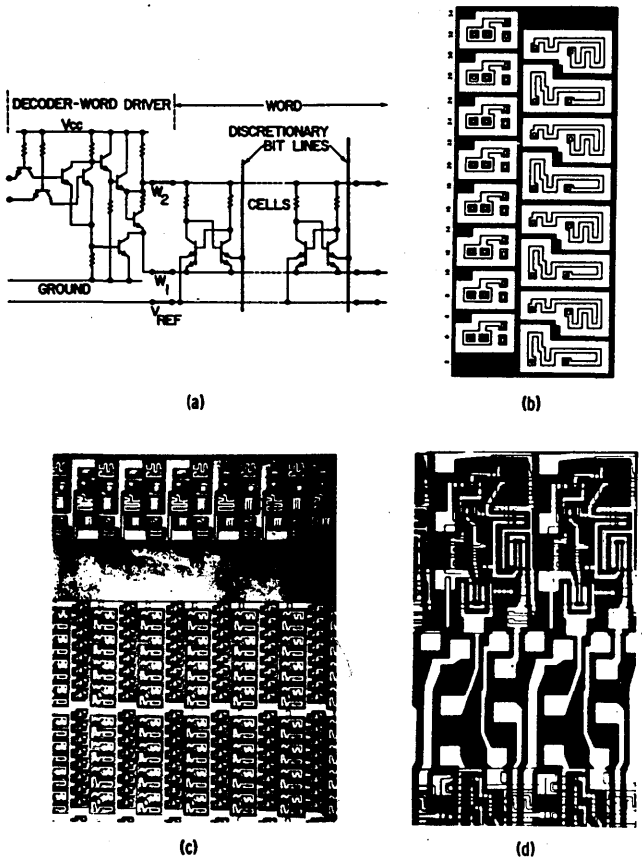


Figure 15—(a) Memory circuits; (b) layout of storage cells, two transistors and two resistors constitute a cell; (c) view of unmetalized memory slice showing decoder-word drivers (top), cells (bottom); d) metalized memory slice decoder-word drivers (top), cells (bottom); and interconnections (middle)

At this stage the slice is probed and the good and bad cells are identified. Figure 16a is a map of a typical slice. From this information a single discretionary mask is designed by a computer, a pattern of which is shown in Figure 16b. Note the relative simplicity of the wiring pattern.

This mask and associated metalization accomplish four discretionary functions. The long horizontal lines pick up 13 bits of the 16 bits in each segment of the word line. Within the horizontal areas between the 16-bit groups, short vertical (or nearly so) lines connect groups of 13-bit word lines. At the top word lines are connected to drivers. Finally, the word and bit lines are connected to pads in the slice edge. Note that these four discretionary functions are accomplished by a single level of metalization even though both horizontal and vertical runs are made. This is accomplished by capitalizing on the regularity of a memory matrix, by employing redundancy, and by some good design work. An article describing this memory

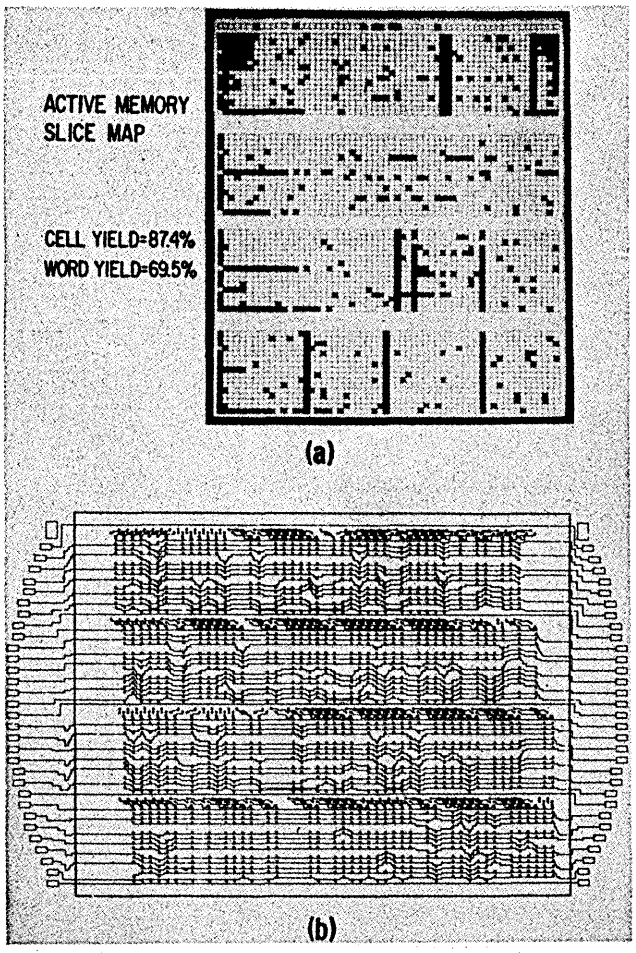


Figure 16—(a) Map of memory slice showing bad (dark) and good cells (b) Discretionary mask drawing for second-level metalization of memory slice

development has been published,[9] and detailed information is given in the contract reports.[2]

The principal technical development required is a two-level metalization technology capable of high yield over a large area. Although this requirement is similar to that for chip IECs discussed above, it is more difficult because larger areas are involved.

B. Customized wiring and discretionary wiring for read-only memory

The basic techniques described above for Read-Write memory are extendible to Read-Only memory. In the Air Force program the same basic slice is used for the Read-Only memory as for the Read-Write memory. The storage bit is simply a transistor of the basic cell shown in Figure 17. The two states are achieved by either permanently connecting the transistor into the matrix, or not.

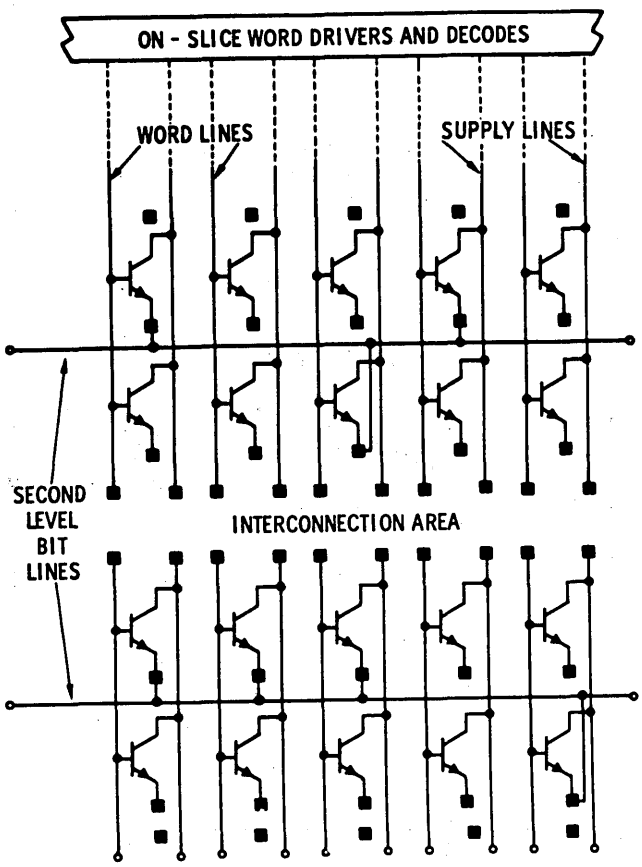


Figure 17—Layout of Read-Only active memory

The first-level metalization is again a fixed pattern (the same for all slices) and serves to connect dc power to the collectors, and word lines to bases for 16-bit groups, as shown in Figure 17. The second-level metalization accomplishes the customized wiring by connecting or not connecting to appropriate emitters according to the customer's software program (horizontal lines of Figure 17). The second-level metalization also accomplishes the discretionary wiring by connecting bit lines only to transistors which have been determined to be good during earlier testing. The second-level metal also connects, as in the case of the Read-Write memory, 13-bit word groups together, word drivers to appropriate word lines, and word and bit lines to pads on the slice edge.

Thus Read-Only memory constitutes a simple but most important example where customized wiring and discretionary wiring are combined.

The Air Force Computer will use Read-Only memory to store the computer program and certain tables of functions. The Read-Write memory will store only data. The volatility problem is avoided since the loss

of power only loses signal data, the Read-Only memory being non-volatile.

C. New memory functions

An active-memory matrix is a function which has not been used extensively in digital systems because of the circuit costs. However, because of the unique features of memory circuits in arrays, the cost of the storage cell may be made a fraction of the cost of a single gate. Scratchpad-memory arrays are now finding numerous applications and batch processing of memories is being investigated. In the Air Force LSI computer system, active-memory arrays will also be used to implement the serial delay function in the video integrator. For this application two logic arrays are required to control three active-memory arrays. This unit will provide the function equivalent to six 608-stage shift registers. A block diagram of this serial delay function is shown in Figure 18.

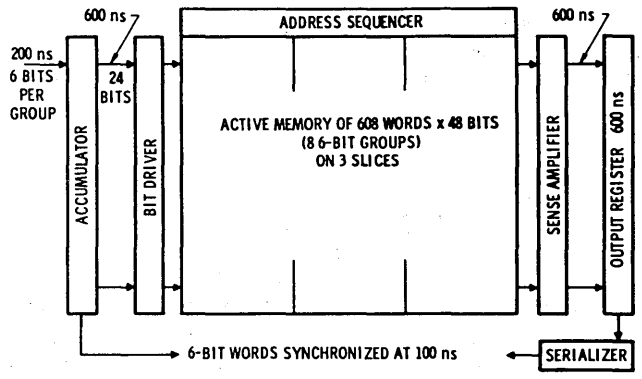


Figure 18—Block diagram of serial delay function active memory

The active memory approach to serial delay functions is more efficient than existing shift-register functions because only the accessed word need dissipate power, and the rest of the memory may be held at a minimum non-select condition. Shift-register units, by comparison, must be active at all times. The disadvantages of the memory approach are the complications and cost of the peripheral logic needed for accumulation, serialization and address sequencing. The minimum size for a serial delay memory is 2000 to 3000 bits.

Another memory function that can be implemented in an active-memory matrix is that of a content-addressable (search) memory. This results because logic can be incorporated in the storage bit.

The author is very optimistic about the use of LSI full-slice technology for memory applications. While our Air Force program goals are for 1024 bits of Read-Write or Read-Only memory, this number should go to 2.5-5 K bits of bipolar memory per slice in the years ahead, and even higher for MOS.

D. Logic program for complexity levels to 250 gates

The Air Force program at TI has a goal of providing customized wiring for logic in addition to the use of discretionary wiring for yield enhancement. Complexity levels of 100-250 logic gates are the program goals. To this end two basic slice types have been defined for logic as shown in Figure 19 (the two memory slices are also shown). Slice (a) provides 881 T^2L gates; the circuit and layout of the gate are shown on Figure 20a and b, respectively. Figure 20c shows the metalized circuit as a part of the entire slice of Figure 20d. Slice (b) in Figure 19 is the slice to which the logic partitioning of Table VI refers.

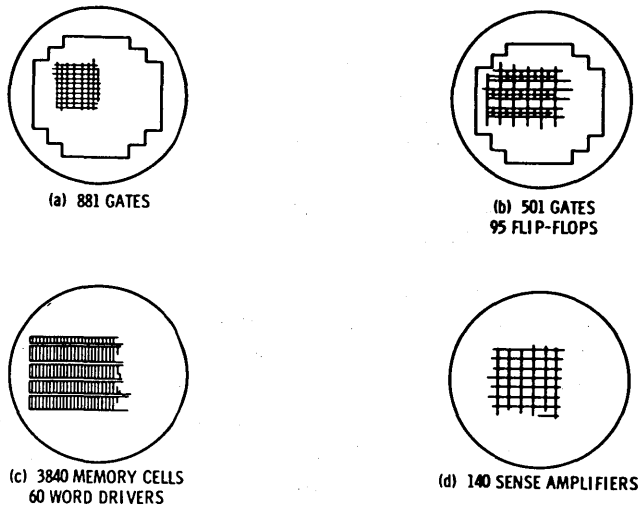


Figure 19—LSI basic slice types: (a) gates for logic functions; (b) gates and flip-flops for logic functions; (c) memory slice of drivers and cells; (d) sense amplifier slice

The customized wiring requirement of the logic portion of the computer has already been discussed in terms of Table VI, where the usage of different slices is tabulated. Our study has shown that the coupling of customized wiring with discretionary wiring for random logic requires three levels of metalization; of these only the first-level metal is a fixed pattern. The slice is probed after first-level metal, and then four discretionary masks are required for each slice to achieve second- and third-level metalization.

Development of this technology requires the development of routing software, low-cost rapid interconnection mask making, and three-level interconnection metalization technology. Considerable progress has been made in each of these areas and is documented in the contract reports[2] and publications.[10]

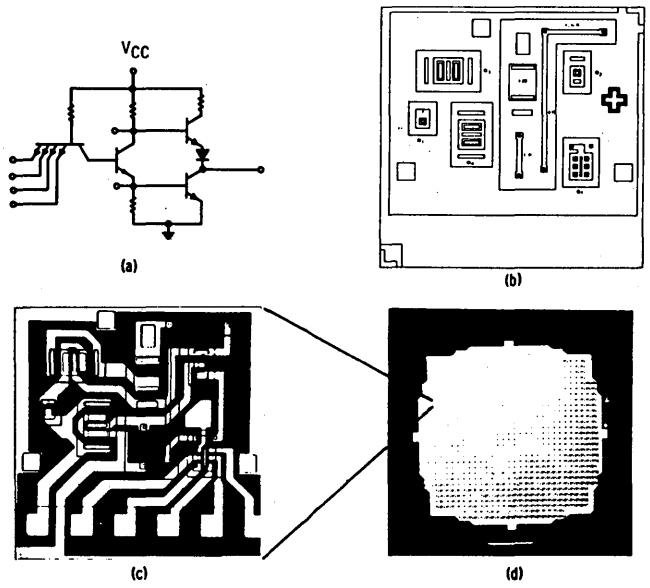


Figure 20—(a) Circuit diagram of LSI gate—this gate is functionally the same as the basic Series 54 gate; (b) layout of LSI gate; (c) expanded view of LSI gate on semiconductor slice; (d) semiconductor slice containing 881 gates

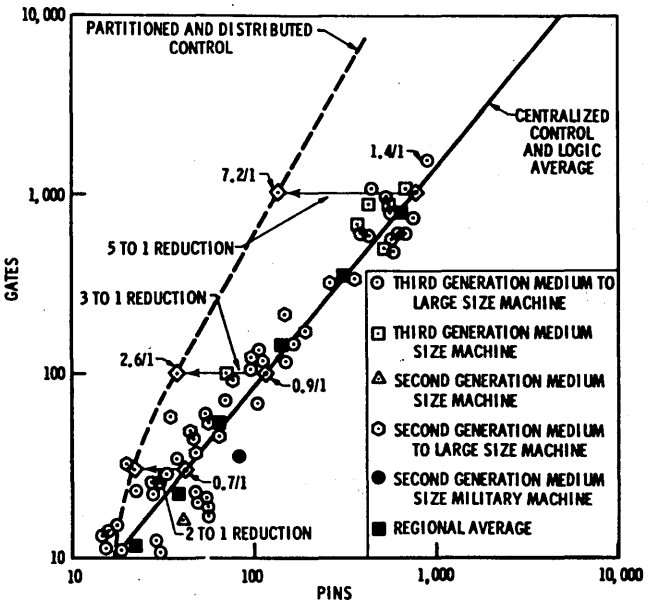


Figure 21—Plot of number of gates versus number of pin connections; the data points relate to the computers as listed in the diagram; the dashed line shows the reduction in pin requirements for partitioned and distributed control as compared with centralized control

E. Logic program for complexity levels 1000-5000 gates

Let us now discuss the prospects for logic function products of very high complexity level (>1000 circuits/slice). One important potential problem is the pin-to-gate ratio. Figure 21 presents data[3,11] from three generations of CPUs that show a linear relationship between gates and pins. Assuming one could put 1000 gates on a slice of silicon, one would still have the prospect of a 1000-pin package. This problem, which relates to partitioning, has received study during the past few years. An important result presented recently[3,11] is summarized on Figure 21. The line titled "partitioned and distributed control" shows that 1000 gates can be accessed with about 150 pins. This is achieved by distributing the control function. In terms of LSI technology this means incorporating control circuitry on the slice of silicon. This has an analogy to the memory function, where by placing address decoding (a control function) on the slice the pin problem is also greatly alleviated (Figure 13).

It is the author's opinion that logic, as well as memory, will utilize the full-slice LSI technology approach for IECs of complexity greater than 1000 gates. However, much work must be done at the systems-architecture level to determine what these products will be. It is quite probable that sufficient regularity will need to be designed into the logic to assure that some of the techniques that have been developed in memory programs will be applicable to logic.

IV. LSI MOS technology

A. Review of MOS program history through development of ratio circuitry

It was recognized early in the MOS development (3 to 4 years ago) that complementary structures (n- and p-channel) could achieve considerably faster switching speeds than single-channel structures. Coupled with this was the attractive feature of nearly zero dc power drain. For this reason one school has devoted a large portion of its effort toward developing a complementary monolithic MOS technology. RCA is the principal industrial firm taking this approach. The Air Force program[1] includes research at RCA[3] to develop MOS memories utilizing complementary technology. RCA has made considerable progress in this effort, and the reader is referred to the firm's reports[3,12] for detailed information.

Another school has taken the approach of restricting the technology to single-channel devices (p-channel in practice) and has investigated circuit innovations as a means of improving over-all systems performance. Their argument has been that the main features of MOS processing technology, namely, the simplicity and the small area required for a device, were to a large extent lost with the use of complementary structures. The same number of process steps are required for complementary MOS as bipolar transistors; they also require isolation diffusions

which waste area just as in the case of bipolars. Leading advocates of this second school have been Philco-Microelectronics' contract[4] with the Air Force[1] involves the development of single-channel MOS-LSI technology.

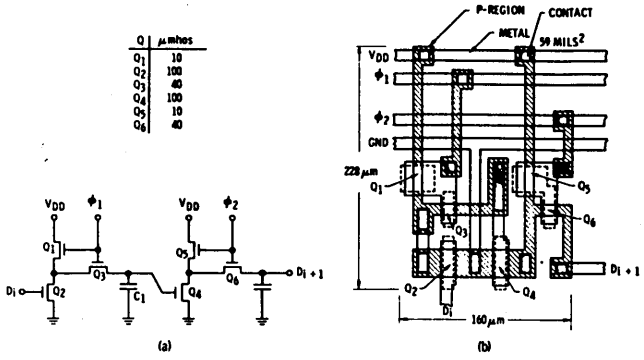


Figure 22—(a) Two-phase ratio MOS circuit diagram; (b) layout of same

Considerable circuit innovation had already been accomplished in the single-channel MOS program as of about a year and a half ago. This served to bring the technology from its original 100 kHz clock rate to 1 MHz clock rates. The main features that were exploited are listed below and illustrated in Figure 22.

(1) The use of an active MOS device as a load resistor which tracks with logic devices.
(2) The ability to turn these load resistors on and off with a clock pulse.
(3) The use of gate capacitance as a temporary storage media.
(4) The use of bilateral properties of MOS structures to transfer charge.

A circuit layout, shown in Figure 22, is contained in 59 mil². This particular arrangement, when integrated, is reported[4] to have one megacycle capability over the full temperature range, or propagation delay of about 200 nanosec. This is called a two-phase ratio shift register—the term ratio recognizing that the g_m of Q_1 is considerably smaller than that of Q_2 (1/10 in the example of Figure 22.) Because of this ratio, the charging time constant is nearly ten times the discharge time constant. This ratio of ten is needed in order to achieve good definition of the 1 and 0 states at dc.

The potential speed advantage of complementary structures results from the fact that the g_m can be made the same (ratioless) and still maintain good definition of logic states. Thus the charging and discharging time constants are equalized and reduced considerably below the ratio type of circuit.

B. Ratioless circuitry—four phase

The important recent development of MOS circuit technology is that methods have been devised whereby "ratioless" circuitry can be achieved while maintaining the use of single-polarity (p-channel) MOS technology. One example of this is the four-phase ratioless shift register, shown in Figure 23, which offers a speed advantage of 10 to 20 over a ratio-type shift register. The phasing is arranged so that there is no dc path to ground—thus achieving the other main advantage of complementary technology, low power.

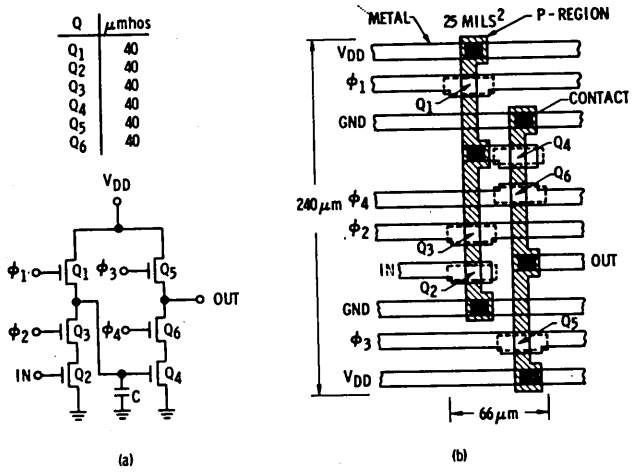


Figure 23—(a) Four-phase ratioless MOS circuit diagram; (b) layout of same

Philco-Microelectronics reports[4,12] a 200 bit four phase shift register in a 90 mil×90 mil area which operates up to 10 MHz with a power dissipation of one to two mW per bit at that frequency, 100μW per bit at one MHz, and 7 μW at 100 kHz. Logic functions can be incorporated into the four-phase system without difficulty. The ratioless principle can be extended to a number of different configurations such as using capacitors as loads. Further information is given in the reports.[4]

In conclusion, significant improvements have been made in MOS circuit techniques the past two years. Using the unique properties of the MOS device and taking advantage of multiphase clocks, it has been possible to design basic building blocks for shift registers and logic with significant performance improvements and size reduction. In 1965, a master-slave MOS flip-flop using straightforward dc gating techniques required 150 mil² of area, operated at 100 kHz, and dissipated 4mW. Today the same function requires only 20 mil², operates at 10 MHz, and

dissipates 2 mW at 10 MHz and only 7 μW at 100 kHz. In both cases, the same device tolerances and photolithographic mask design rules are used.

C. Technological factors governing MOS IEC performance

MOS performance improvements have been discussed above. Technology improvements are also expected.[15]

The intrinsic g_m/c cutoff frequency of a voltage-controlled device such as a MOSFET suggests it to be capable of bandwidths in the order of 1.5 GHz or better. This translated into 10-90% rise times yields 0.24 ns. Assuming 0.25 ns for each rise, fall, and pulse duration a typical MOS would toggle at 1 GHz. However, one does not observe this speed, today's limit being about 10 MHz as discussed above.

The primary speed limitations in MOS IECs are:

(1) Low device transconductance (g_m)
(2) Stray circuit capacitance
(3) High voltage levels
(4) Inadequate output buffers

The first two limit speed because of the MOSFET's inability to drive stray circuit capacitance at a high rate. This is because the MOSFET has a relatively low transconductance per unit area when compared with a bipolar device. This point has been discussed in other literature.[6,16]

To examine how the speed-power performance may be improved, consider the following basic equations for MOS devices:

$$g_m = \mu(\epsilon/t)\,(W/L)\,(V_g - V_T)$$
$$C_s = KA$$
$$\tau = (C_s/g_m)$$
$$P_{AV} = CV^2f$$

where μ is mobility, ϵ and t are the dielectric constant and thickness of the gate insulator respectively, W/L is the width-to-length ratio of the MOS device, C_s is the stray capacitance, and V is voltage.

Examination of these equations suggests the following areas for improvements in MOS performance.

(1) Higher mobility will improve performance—this is one reason that n-channel devices are still of potential interest.

(2) Increasing the ϵ/t factor increases g_m faster than the total C_s, thus reducing τ. Higher dielectric constant gate insulators such as silicon nitride (Si_3N_4) and alumina (Al_2O_3) are being studied for this reason.

(3) The area, A, can be reduced by improved photomasking techniques, thus reducing C_s and improving τ.

(4) A reduction in the voltage level swings for both clock pulses and logic levels will improve speed-power performance since P~to V^2 while g_m~V. Thus power will decrease faster than gain for a voltage reduction. The present clock levels of 24 volts and logic levels of 12 volts severely limit the ultimate speed capability of MOS IECs. Research to better understand and control MOS threshold voltages is under way in numerous laboratories, and significant improvements can be expected.

(5) The problem of inadequate buffers arises because, while stray capacitances can be kept to relatively low values (~1 picofarad) while on the chip, the capacitances encountered when going off the chip are much larger (~20 picofarad). As a result, output buffers are generally slower than internal nodes. Enlarging buffers to obtain increased g_m is not an entirely satisfactory solution because of the large area required. New types of buffers must be considered such as: pre-charged, quasi-complementary, and MOS bipolar combinations.

In conclusion, the combined effect of innovations in MOS circuit design, coupled with further improvements in MOS device technology, will result in further improvements in the performance of MOS IECs.

D. MOS custom products

A number of MOS standard products are already on the market, but these are rather limited in scope, being principally shift registers. It is generally agreed that custom designs will be very important to MOS development. One approach to customizing is the master-slice, similar to that described above for bipolar technology. Fairchild, who is following this approach, provides an array of master MOS cells, and the customizing occurs at the first and second level of metalization. The chips are 6400 mil^2 in area and contain 80 three-input NAND gates, an effective area of 80 mil^2/gate.

The "family of cells" is the second approach to customization and is being pursued by Philco-Microelectronics, General Instruments, Texas Instruments, and others. This approach capitalizes on the simplicity of the MOS process, coupled with the high circuit-density potential of MOS. Here one defines a family of basic circuits (cells) such as shift-register bits, NOR gates, and flip-flops. These cells are designed and laid out in an optimized, compatible fashion. Using computer-aided design, artwork is generated which places and interconnects these cells to achieve the customer's functional requirement. Masks for both diffusion and metalization are then made from this artwork. While this is more costly and time consuming than the use of only metalization masks, advocates believe that the better utilization of chip area will more than compensate for this.

E. Full-slice MOS technology

MOS LSI technology has been directed almost entirely toward chip IECs, principally because of the high complexity level achievable in small chip areas. However, there is interesting potential for full-slice MOS LSI technology, such as memory IECs. Studies at TI show that single-channel MOS memory cells will require about half the area of bipolar cells; namely, 50-60 mil^2. One can visualize very large memory functions on a single slice of silicon; 5 to 10 K bits are within reason. The use of redundancy and discretionary wiring could be applied similarly to the above description on bipolar memory. Such memory should compete very favorably against magnetic-core technology for moderate speed (1-2 μsec) main-frame memory.

V. LSI hybrid technology

We have discussed the chip and the full-slice LSI technologies; the third well-defined technology is hybrid. Here one integrates to a certain level on the chip, and then employs film technology for interconnecting the chips within a package. Figure 2 pictorially summarizes the three basic LSI technologies.

Hybrid technology is being developed by a number of companies, Bell Labs being particularly strong. They are employing beam-lead techniques for low-cost assembly, silicon nitride for passivation to achieve low-cost plastic packaging, and thin films for interconnection and certain passive components. A report[14] on this work was given recently.

TI is developing nanosec logic IECs by the use of hybrid technology. The approach is to push to very high speed (1-2 ns) chip technology, sacrificing complexity levels per chip. The present goal is 20-25 ECL circuits per chip. In order to achieve circuit densities of 100 circuits/inch^2, 4 to 5 chips will be interconnected using thin-film technology.

VI. Cost, performance, and reliability versus complexity considerations for LSI technologies

Of ultimate concern to both manufacturers and users of IECs is the choice of a technology that will result in the most favorable cost structure for a given system application. It is clear from the discussion in Section II that the complexity levels for minimum systems cost have shifted from one circuit per package in 1962-63, to four circuits per package in 1965-66 to upwards of 30-40 circuits per package in 1967-68.

A forecast of the complexity level of minimum cost as well as the technologies required to achieve minimum cost for specific requirements is a complex subject. Not only basic costs, but also performance, reliability, and other factors such as custom versus standard products, must be considered. While detailed answers to many questions are not yet available, considerable progress has been made during the past year. We discuss some aspects of this work in the following sections.

A. Cost-complexity studies

1. Cost Complexity for Standard Bipolar T^{2}L IECs up to 250 Logic Circuits, 500 Memory Bits Complexity

A study of the costs of standard T^{2}L IECs has been made, considering both the chip and the full-slice technologies. The study of yield versus area as discussed in Section II was a major input. Assuming that the yield will improve as forecasted in the yield versus area study, the corresponding cost-complexity study forecasts that the chip (100% yield) technology will result in lower costs than those which can be achieved by the full-slice (discretionary wiring) technology. This forecast assumes that chips up to $\frac{1}{4} \times \frac{1}{4}$ in^2 will be produced at reasonable yields and also that there will be sufficient production volume to maintain good processing conditions. The full-slice approach utilizes too much silicon area and bears the additional cost of generating individual masks for each slice to be competitive in this complexity level for standard products of high volume production.

2. Cost Complexity for Custom Bipolar T^{2}L IECs for Complexity Levels up to 250 Logic Circuits

As discussed in Section II, the complexity level of 10-250 logic circuits is particularly important for custom IECs. The cost-complexity analysis is much more difficult for this problem, because in addition to production-cost considerations one must consider the engineering and tooling costs for relatively small-quantity requirements. Let us briefly review the two basic approaches to this requirement discussed earlier.

The chip-technology approach must consider the problem in the light of generating three custom masks: first-level metalization, insulation, and second-level metalization. Material must be processed with these custom masks, and for complexity levels 100-250 circuits, yield cannot be expected to be high, particularly for short runs. However, the custom masks are the same for each slice processed and thus are a fixed engineering and tooling cost.

The competing approach is the combination discretionary-wired custom-wired logic technology described in Section III. Here the design costs for the custom masks are absorbed in the software costs of the discretionary masks. However, these costs occur

for each slice processed so must be considered a production cost. Three levels of metalization are required, one fixed pattern, and two discretionary patterns. Also, the yield of the two-level discretionary-metalization technology must be quite high because the circuit testing, software costs, and discretionary mask costs are incurred prior to this stage of processing.

We should be able to define the roles of the two technologies better a year from now since experimental data on the two approaches are being generated only now.

3. Cost Complexity for Standard and Custom Memory Bipolar IECs for Complexity Levels of 1000-5000 Bits

Three semiconductor technologies must be compared for this important application area: chip, hybrid, and full-slice. In addition, a comprehensive analysis should include competing magnetic technologies, but we will restrict our comments to the semiconductor technologies. Let us consider a complete memory function with associated address-decode, driving and sensing circuitry as the example for discussion.

The chip technology costs must include not only the costs of processing the chips and assembling them into packages, but also the costs of assembling the packages onto multilayer boards, and the costs of multilayer circuit boards.

The hybrid technology must undergo a similar analysis, the only difference being that the chips are interconnected and assembled with film technology into a higher-level package which contains the memory function. Key questions yet to be answered here are the yield for the film interconnection technology, and to what degree repairability is possible.

Full-slice technology has been described in Section III for this application. Here the discretionary wiring involves only the final interconnection mask, and the software costs for generating this mask are minimal because of the regularity of the problem. The major unknown in this approach is the yield of second-level metalization. We are processing material in substantial quantities for the Air Force memory[2] and should have the answer to this question in the near future.

While we lack precise data on these three approaches, the author's opinion is that the discretionary wiring, full-slice technology will provide an economical and effective solution to this requirement area. The ease of coupling customized wiring with discretionary wiring makes this approach promising for customized Read-Only memory as well as the standard product Read-Write memory.

4. General Comments on Cost Complexity

We have attempted by discussion of three specific application areas to give the reader some insight into the general problem of cost-complexity. It should be emphasized that there are no pat answers to these questions; specific problems require specific study, and there is no solution for this other than engineering studies between component suppliers and system houses. There is a broad technology base from which to choose, and careful study should yield good solutions to specific problems.

B. Performance-complexity considerations

1. Low-Speed Applications (1 MHz, or Less, Clock Rates)

The area advantage of p-channel MOS technology over bipolar is now unquestioned. A detailed report on this was recently published.[16] For new designs where interfacing problems do not dominate, MOS is recommended as having a cost advantage over bipolar.

2. Very High Performance Applications (Less Than 2-3 ns)

Non-saturated bipolar technology as represented by ECL circuitry stands relatively unopposed. The main question lies in the degree of integration on the chip, and the corresponding use or non-use of hybrid interconnection technology. As mentioned earlier, at TI advanced device structures are being employed on the chip to achieve greater circuit speed. Because of yield considerations this requires holding the level of integration on the chip to relatively modest levels (~25 circuits per chip). Thin-flim interconnection technology will be utilized to achieve circuit densities of ~100 circuits per in^2. Variations of this approach are being used in super-machine projects now under way.

3. Moderate Performance Applications (5-100 ns)

This range of applications appeared to be the unquestioned domain of bipolar technology until the advent of the 4-phase ratioless MOS technology described in Section IV. At the top end of the speed range (5-10 ns) this is still the case, the competition being between saturated and non-saturated bipolar technologies, with ECL probably dominating if speed is the main consideration.

The competition for the lower end of this speed range, 50-100 ns, must consider the ratioless MOS as well as the various forms of saturated bipolar (T^2L, DTL). Over-all cost for system through-put could favor MOS because of the higher complexity levels achievable within chip areas.

For the medium portion of the speed spectrum (10-50 ns) bipolar technology should dominate, with the choice being between T^2L and DTL.

C. Reliability-complexity considerations

The Air Force contract has as a specific goal the achievement of at least a tenfold increase in reliability through LSI technology as compared with present-day integrated circuits. The background to this goal is that properly designed and processed semiconductor products result in system failure modes that occur principally at bonding pads, solder joints, etc., and not on the semiconductor chip itself. While it is dangerous to over-simplify, the argument follows that increasing the complexity level on the semiconductor chip or slice, while decreasing the number of bonds and external interconnections, should result in an over-all increase in systems reliability. Under this assumption, the reliability-complexity analysis favors technologies that have higher complexity levels on the semiconductor chip or slice.

D. Over-all cost, performance and reliability-complexity considerations

In a given systems application consideration must be given to the three factors—cost, performance, and reliability versus complexity, in addition to numerous other factors such as software-hardware tradeoffs. Since the relative weight given to the particular factor depends on the particular system, no general figure of merit versus complexity seems appropriate. However, significant data now exist or are being developed, assuring that intelligent studies can be made of these basic system questions.

ACKNOWLEDGMENT

The author gratefully acknowledges the contributions of the many authors to whose work he has referred. He also thanks the Air Force for permission to use data from their contract reports. Finally, to his many colleagues at Texas Instruments—including particularly Richard Abraham, Willis Adcock, David Chung, Robert Crawford, Roger Dunn, Robert Hibbard, G. E. Jeansonne, Jack Kilby, Jay Lathrop, Charles Phipps, and Leonard Short—he gratefully acknowledges many helpful discussions.

REFERENCES

1 *Large scale integrated circuit arrays*
Air Force Contract nos. AF 33(615)-3546, 3491, 3620 Technical Monitor: Robert Werner Air Force Avionics Laboratory Wright-Patterson Air Force Base Ohio

2 *Large scale integrated circuit arrays*
Air Force Contract with Texas Instruments Incorporated Contract No. AF 33(615)-3546 Project Supervisor, Jay W. Lathrop Five quarterly interim reports issued through May 1967

3 Air Force Contract with Radio Corporation of America Contract no. AF 33(615)-3491 Project Supervisor, G. B. Herzog Five quarterly interim reports issued through May 1967

4 *Large scale integrated circuit arrays*
Air Force Contract with Philco-Microelectronics Division Contract no. AF 33(615)-3620 Project Supervisor: Donald Farina Five quarterly interim reports issued through May 1967.

5 R L PETRITZ
Large scale integration technology
Trans. Met. Soc. AIME, vol. 236 pp. 235-49 1966

6 Ibid.
Technological foundations and future directions of large-scale integrated electronics
AFIPS Conference Proceedings 1966 Fall Joint Computer Conference pp. 65-87 November 8-10, 1966

7 R B SEEDS
Yields, economics and logistic models for complex digital arrays
Conference Record 1967 IEEE International Convention and Exhibition March 20-23, 1967

8 E G FUBINI M G SMITH
Limitations in solid state technology
IEEE Spectrum, pp. 55-59 May 1967 also Electronics Magazine pp 130-143 Februray 20 1967 with W A Notz E Schischa and J L Smith

9 R S DUNN G E JEANSONNE
Active memory design using discretionary wiring for LSI
1967 International Solid-State Circuits Conference Digest pp 48-49 February 1967 also Electronics Magazine pp 143-156 February 20 1967 with M Canning

10 J W LATHROP
Discretionary wiring approach to large scale integration
Western Electronics, Show and Convention 1967 also 1966

11 H R BEELITZ H S MULLER R J LINHARDT R D SIDMAN
Partitioning for large scale integration
International Solid-State Circuits Conference Digest pp 50-57 February 1967

12 A K RAPP L P WENNICK H BORKAN K R KELLER
Complementary-MOS integrated binary computer
International Solid-State Circuits Conference Digest pp 52-53 February 1967

13 J KAPP E de ATLEY
Using four-phase IC logic
Electronic Design pp 62-66 1 April 1967

14 T R FINCH
LSI—Digital electronics
International Solid-State Circuits Conference Digest pp. 32-33 February 1967

15 R H CRAWFORD
Practical considerations in the speed limitation of MOSFET ICs
NEREM 1967

16 R M WARNER JR
A comparison of MOS and bipolar integrated circuits
1966 NEREM Record pp 68-69 also IEEE Spectrum pp 50-58 June 1967

Large–scale integration from the user's point of view

by M. G. SMITH and W. A. NOTZ
IBM Watson Research Center
Yorktown Heights, New York

INTRODUCTION

The potential LSI user views LSI promise with a great deal of anticipation, but LSI problems with some trepidation. Obviously, he hopes for breakthroughs to relieve the strain of having to squeeze the last bit of cost or performance from the existing technological approaches—and of having to contend with the added hardware and software problems fostered by the need to improve his product only through system complexity. There are, in fact, very few things the system designer can do that have the impact of a significant technology advance in cost or performance.

LSI could be this sought-after breakthrough, even though some may see it as more of an evolution from low-level integrated circuits. Certainly, the basic manufacturing costs should eventually be significantly lower than today's circuit costs—and we should also see improved performance, increased reliability, smaller size, lower power, and easier serviceability. On the other hand, there are possibly, very significant negating factors, such as, high part number costs, long turn-around times, and difficult test and specification problems.

There would have to be many points of view concerning LSI from users, subject at least to the user's application and internal capabilities. Is it a large or small system; is there a high or low market volume expected; is it a high or low performance application; are there particularly sensitive cost, power or reliability requirements; what facilities does the user have at his disposal; and does the user plan to buy or make his components? Obviously, LSI is not equally attractive to all these users; particularly if we are not allowed to deal in an "ultimate" time scale.

Logic costs

Will LSI mean lower cost systems in the context which we know systems today? Consider a hypothetical CPU (Figure 1) which is somewhat representative of an intermediate-sized cost-performance system today. The example is chosen to be favorable to LSI. (In this example, we assume that the "LSI-equivalent" system is identical in function and performance although this may not be the way we would want to use LSI.) Reductions are assumed due to LSI use in the logic and special memory areas, but not in main memory. These costs assume very substantial reduction in silicon processing cost, i.e., more than an order of magnitude; such that logic circuits, for example, packaged to the module level, cost an order of magnitude less. Thus, if circuits cost, say, 50 cents today; then they would be about five cents apiece in our "LSI-equivalent" machine.

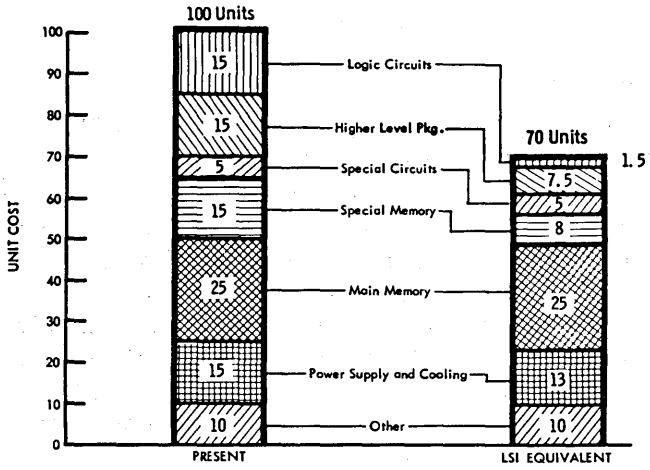


Figure 1—Hypothetical CPU costs

In overall CPU costs, we have saved 30%, a tidy sum no doubt; but far from the more than one order of magnitude in original silicon costs. However, consider this CPU in the hypothetical system reflected by the cost of Figure 2. (This system is a near minimal configuration and does not load the cost with a great deal of conventional I/O hardware, again favoring the LSI cost savings.) Our percentage saving has now been reduced to 18%.

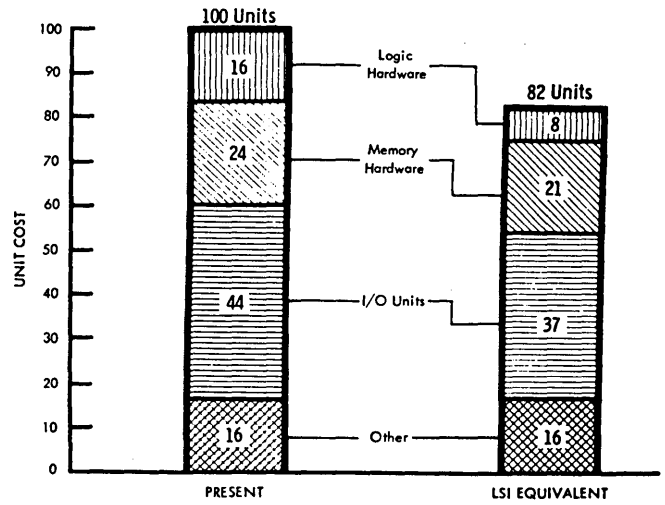


Figure 2—Hypothetical system costs

More function per dollar

What should we do with this saving? The key point here is that it could be turned back in for significantly more hardware capability; for example, about three times as much logic and twice as much special memory. Of course, there is still more of this story, as indicated in Figure 3, a hypothetical cost of ownership. (Certain costs such as training personnel are omitted.) Here our expectant user, filled with the anticipation of an order of magnitude decrease in cost, may have to settle for about 7%; and, as a matter of fact, many system examples would look worse.

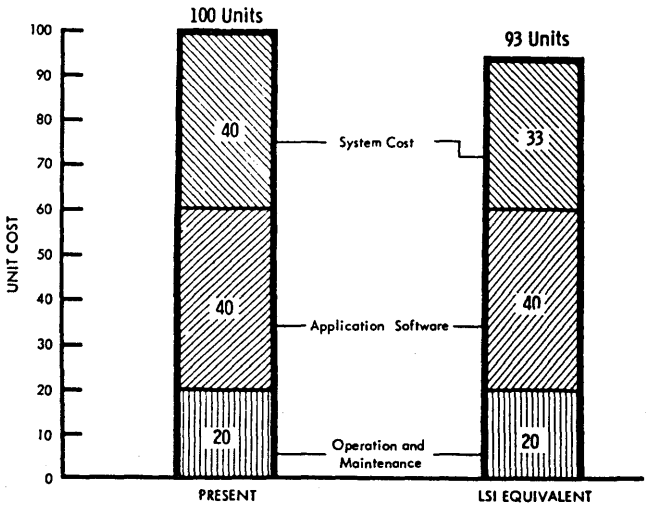


Figure 3—Hypothetical cost of ownership

A major cost to the system user is the cost of applications programming. If LSI hardware costs can be low enough, surely, here is a most challenging application, and clearly, we should be able to add significantly more circuitry to accomplish many functions which previously were either available only in the most expensive machines, or prohibitively expensive all together.

Some examples of functional expansion we would naturally consider are as follows. In the central processor LSI might be used to carry out more micro-operations per instruction; address more operands per instruction; control more levels of look-ahead; and provide both repetition and more variety in the types of functions to be executed. In system control, LSI might provide greater system availability through error detection, error correction, instruction retry, reconfiguration to bypass faulty units, and fault diagnosis; more sophisticated interrupt facilities; more levels of memory protection; and concurrent access to independent memory units within more complex program constraints. In system memory, LSI might provide additional fast local memory for operands and addresses; improved address transformation capability; content-addressable memory; and special fast program status tables. In system input-output, LSI might provide more channels; improved interlacing of concurrent input-output operations with automatic memory protection features; and more sophisticated pre- and post-processing of data and instructions to relieve the central processor of these tasks.

Special organizations

Certain special machine organizations and functional memories have considerable part repeatability and will be well suited to LSI from a fabrication standpoint. The machine organizations and functional memories referenced here have in common the characteristic that they consist of a number of identical data processing units or cells that are capable of concurrent operation. Otherwise, they can be quite different.

From a hardware point of view, they may vary from a few gates in the simplest associative memory cell or cutpoint logic cell[1] up to thousands of gates per processing unit in machines such as ILLIAC IV.[2] Included are the SOLOMON[3] and Holland[4,5] organizations.

Most of the special organizations of large arrays of identical computing units have been shown to be capable of solving any kind of data processing problem but have only been shown to be efficient relative to more conventional single processor organization in selected applications.

What then should the system organization be for LSI? Should we stick with existing organizations which characteristically leads to many part numbers, or should we consider special organizations around a universal part number? The choice here is certainly very much a matter of costs, and even more, a matter of where the costs are incurred. For example, if circuits cost nothing, but

part number charges are very high, the special organizations will be greatly enhanced. On the other hand, if part number costs (including turn-around time penalties) are negligible relative to the volume, we might expect that today's system organizations will prevail or at least will not be altered due to LSI alone.

Some studies have attempted to show the proliferation of parts economically possible if the number of part-numbers could be substantially reduced (assuming a range of system quantities, projected part number costs and other pertinent parameters).[6] Implicitly, identical system functions are assumed, although this is not likely. This proliferation is small for system quantities in the thousands and extremely high for the one-of-a-kind system. However, the LSI implementation of few-of-a-kind systems actually will be bounded by other, low-level-integration implementation means and costs. Thus, in the foreseeable future, only a small factor, generally between one and four times the amount of logic can be expended to create a one-part-number system from a many-part-number system. Generally, this is not adequate to implement the original function. This also implies that special organizations, with extensive use of repetitive logic (and memory) functions, will still have to be justified by what they can do differently and by significant improvements in performance. However, these systems may now be economically possible, at least, with LSI.

Volumes and part numbers

The importance of circuit volume and volume per part number is amplified in LSI. Aside from memory, where do we find LSI-compatible volumes? First, the total circuit volume is often higher for small machines than for the large central processing units (we include in the small-machine-category terminals and other peripheral equipment for larger machines as well). Also, the small machine, because it is usually sold in larger quantities, generally offers us more parts per LSI part number than does the conventional large machine (Figure 4). The average number of parts/part number is plotted for a plausible distribution of higher volume systems in each size category.

A "normal" part number repeatability in a total system is illustrated in Figure 5. Obviously it can be very high in memory, both within a system and in some cases across system types. Registers, of course, fall into a similar category. Certain portions of the control can also be fast-read, slow-write memory. Much of the logic, however, will be unique to the system. In the area of I/O and remote terminals, there is little repeatability within units, but many such units are employed within the system and across the system types.

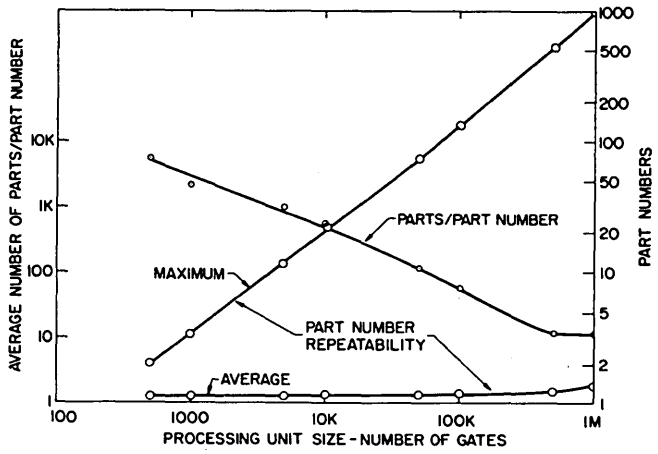


Figure 4—Plausible part number quantities and distributions for "conventional" systems (integration level, 100 gates)

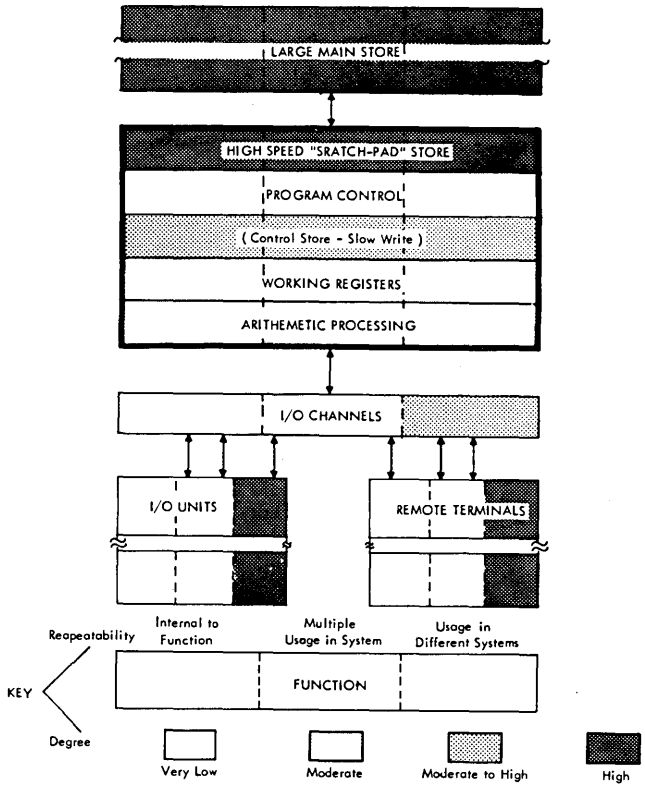


Figure 5—Part repeatability

Manufacturing and implementation costs

Most users must really believe they will have low costs before they will put much time and money into devising schemes to proliferate components in a system. Thus, we should expect that many early users will be timid. Even accepting that the raw manufacturing costs will be low, how much will it cost to generate an LSI part number? It is not unusual today for the mask set

alone to cost $5000 for one part number, assuming a 100-circuit integration level. If the machine size is modest and if it has a high market volume, we could use many manual techniques. Thus, these small, high volume systems or subsystems, along with memory applications, give us a place to start and afford an evolution to more extensive applications as the problems are solved. However, at present costs (Figure 6), the very small-volume machine would require more just for the LSI masks than for implementing the logic hardware by other means—and we have not included the LSI costs for partitioning, placement, wire routing, testing, and an unprecedented amount of interface time; easily another $5,000, today. Of course, under these circumstances, the small-volume system user will have to be motivated by something other than cost *per se,* if he uses LSI.

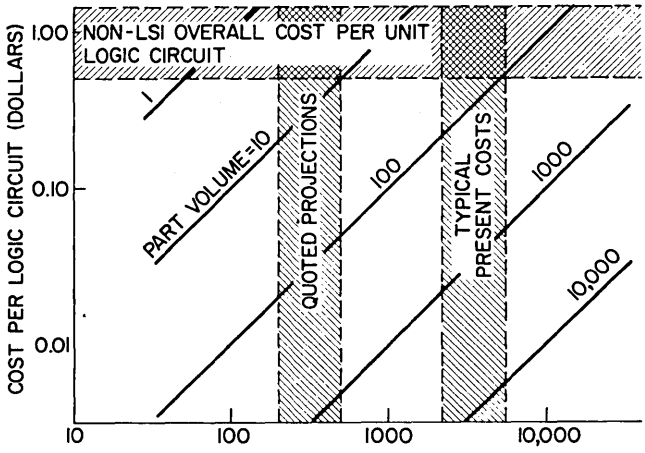


Figure 6—Mask generation cost (dollars)

However, an even greater problem exists for some who can get by all the previous hurdles. This is the delay which may be encountered by the potentially long turnaround times in a system environment already well-known for its high incidence of change; compounded by a much higher susceptibility of error, and costly errors potentially engendered by LSI. Can we live with this? Some system designers have speculated that the development of LSI machines would take 50% to 100% longer. If so, this is a significant loss of investment and sales or revenues. Add to that, delays due to changes after the product is released, and losses after a heavy investment in inventory, and the systems man could have paid many times the amount per circuit in a non-LSI form. In addition, the old problem of early technological obsolescence is worsened.

Implementation aids

We are all familiar with the major elements in these implementation procedures (Figures 7 and 8). Obviously, we cannot eliminate many, if any, of these functions; so what can be done to cut the operating times

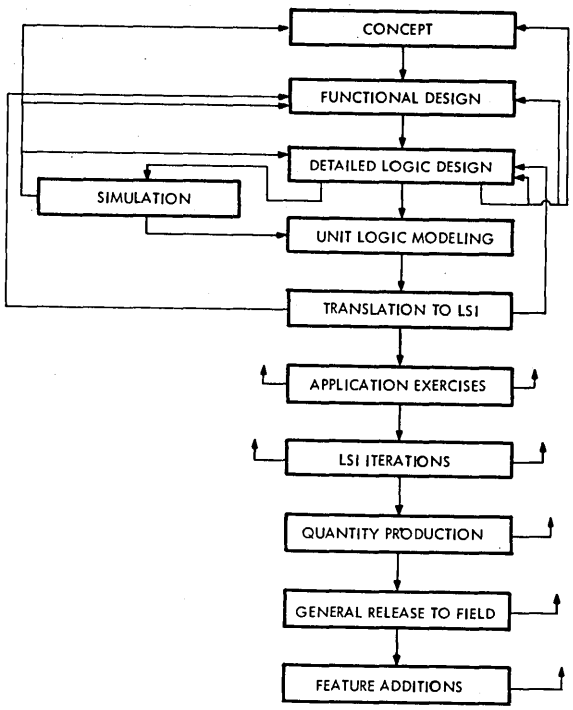


Figure 7—Typical small machine development and manufacturing sequence

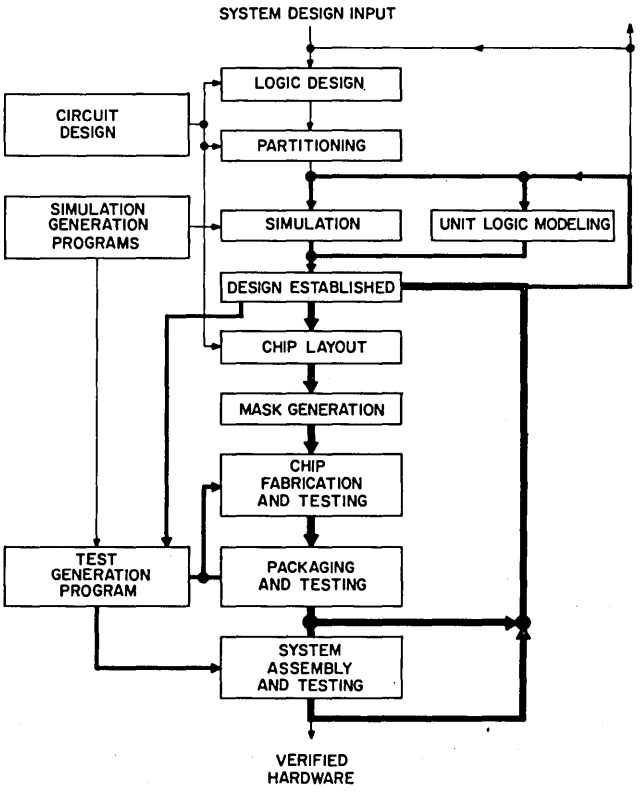


Figure 8—Development-hardware implementation

and costs, and the incidence of iterations? Fortunately, some of these functions are not in the main iteration loops and some of the costs involved are not unique to a particular part number, i.e., they can be shared over a range of part numbers or applications. However, we need much more.

Obviously, we must question both the manufacturer and the user in this area. Can some of these functions be gotten out of the main stream? Can operations be paralleled? Can we collect some function together so that the product spends a minimum amount of time in "in" and "out" baskets? Even while "in process," much time is lost in queuing.

In addition, can we assure that iterations will not take nearly as much time as the first time through? Can the hardware design impact this? That is, should we commit to master-slice approach with only a final metalization; or if we have a very high volume item, should we use a master-slice in development and a more efficient *ad hoc* layout in production? What are the trade-off's?

The user generally likes to construct the system (or a significant part, at least) as an integral part of design procedure. Should we also model hardware in unit logic parts in advance of generating the LSI parts? Also, we have been able to make hardware changes easily in the past, and it has become a way of life. This will have to change.

As indicated, not all the functions have to be in the main iterative loops and, in fact, it is really these loops which the user worries about the most, namely, the loops which he may have to go through if he desires (or must) make changes. Perhaps this is because the total design sequence may be measured in many months, if not years; while the iterations need to be measured in days or weeks. Certainly, if the situation is not significantly worsened by LSI, we can use LSI. Within this context, then, what implementation procedures should we have for LSI? Figure 8 represents the rudiments of such a system, and although iterating loops can take place almost anywhere, the main iterating paths are as indicated. Notice that by implication, we are only considering minor changes. Hopefully, our detailed logic design simulation and unit logic modeling procedures will have proven the basic concepts and functional designs. Notice too, that the circuit design and partitioning are not part of the principal loops. (Conceptually, we could envision circuit design as an integral part of the layout design, adjusting the circuit parameters somewhat, due to the variations in capacitance loading, for example.) Generally the circuit (or circuit family) can be specified along with the necessary design and layout constraints and these constraints can serve as boundary conditions in the layout. Partitioning is not part of the iterative loop if changes are minor. If chips, susceptible to change, are designed leaving some extra chip area and I/O pins, then chips could be changed without impacting the basic partitioning. Of course, each case would need to be examined separately and, with some ingenuity, "fixes" might be made using some special adjunct circuitry.

The major iterating loops are indicated in Figure 8 by the heaviest lines. These loops really reflect that many of the changes may be due to errors in implementing the logic or they are minor enough that a larger loop can be by-passed. However, we should really have fast iterating loops including simulation for the more significant changes we will surely encounter. Note that the emphasis continues to be on iterations and iterations can often be shortened, particularly when using automated design methods.

Automated techniques

Obviously, if we are to experience the dramatic increases in the number and sizes of LSI systems, we must offset the enormously increased burden of labor and time on detailed logic designs, placements, layouts, mask generation, testing programs, and documentation, by highly automated means. In turn, if we are to develop costly automated means to reduce these handling times and costs, we obviously have to develop "master" programs and facilities which can serve many part numbers and system applications. However, it is difficult to see both fast turn-around and low cost in our part-number development cycle. We look to a facility, such as that in Figure 9, to provide us with the necessary flexibility and turn-around times, at least. Functionally, this is a part of most of the iteration loops and, in present-day circumstances, it is notoriously unpredictable, time consuming and costly. Does this mean a special high-speed development facility apart from the manufacturing facility? After all, the manufacturing facility may not be optimized for both fast response and low cost; but an added facility optimized for fast turn-around time may be an expensive addition to our part number costs, since such a facility may have a very low through-put. It could cost several hundred thousand dollars and require a comparable investment in people per year. Certainly, it is not difficult to see a few thousand dollars per part number, and perhaps much more, expended in this facility alone. Perhaps, then, if we are to truly have low-cost part number generation, we will have to envision this facility as a special subset of a larger complex—but we must be certain that this does not seriously jeopardize the turn-around time.

What is being done in the industry in some of these key problem areas? One of the more sophisticated approaches, and actually the earliest truly LSI approach

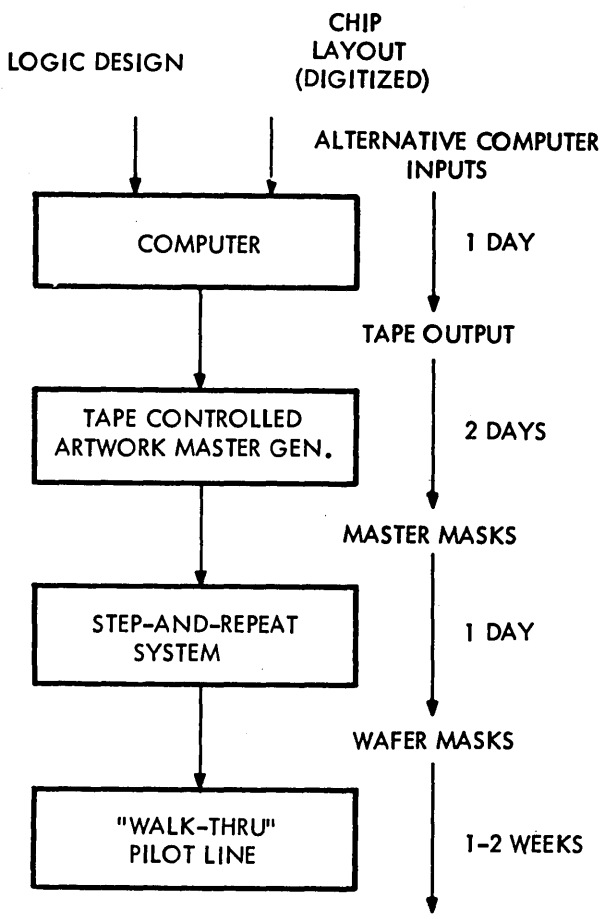


Figure 9—Fast turn-around development facility

is the programmed interconnection pattern (IBM)[7,8] or discretionary wiring (T.I.)[9,10] approach. Circuits are pre-tested and interconnected on a wafer. Interconnection patterns are determined by the computer which in turn guides a light beam directly on the wafer (IBM) or a cathode-ray tube beam to produce a film mask (T.I.). Extremely fast wiring programs have been generated, costing only a small fraction of a cent per circuit, although at some expense in the silicon area (the computer speed is greatly enhanced if the structure is tailored for the computer and if plenty of wiring space is available). Thus, while these schemes may have lower densities than the *ad hoc* fixed pattern approaches, they still may be competitive subject to the various yield and density factors, and they may be useful in lower volume cases. A key point here is that the rudiments of a system exist today to convert logic diagram into final mask, whether it be for the discretionary or fixed pattern approaches. When applied to fixed pattern approaches, which place a high premium upon maximizing the silicon area utilization, the problem is much tougher. (This accounts for some of the apparent discrepancies in the status reports concerning automated "wiring" approaches.)

Mask-making

What are some of the approaches being proposed for the standard fixed pattern approaches? A number of companies have reported efforts recently, including Fairchild,[11] Motorola,[12] and IBM.[13,14] While not all of the details are common to all the parties indicated, in general, each approach provides for both computer generated layouts and a manual input, at least for iterations. Some of these systems provide a graphic console for designer-computer iteration. The output of some of these systems is a rubylith master; at least one uses photographic plates. All of the systems reported are using reduction and step-and-repeat operation to generate the final masks.

An interesting variation recently reported at IBM,[13,14] combines computer and manual layout techniques to provide a very effective mask-making capability. The procedure begins with a sketch of the basic chip format which is also stored in the computer. The computer also stores frequently used structures or patterns of varying complexity which can be used without repeating the detailed coordinates. The designer sketches in his interconnections and devices and key coordinates are entered onto punched cards. From this sketchy information, the computer is able to generate the complete set of several masks, although the designer prepared only one sketch. The program also permits rapid alterations in the masks with the change of a few cards—and this scheme, if combined with a graphic terminal, would permit even faster alterations. It is clear these schemes give excellent flexibility and reduce the layout and mask-making costs significantly; but it does not replace the need for a totally automated system, which is both economical and efficient in the use of silicon area, particularly for low systems-volume situations. Excellent progress is being made in this area.

Testing

Testing continues to be a significant problem, not only because every part has to be tested, but because effective test generation programs are difficult to construct. A number of persons have attempted to illustrate the enormity of the problem in pointing out that at least 2^n variation is possible, where n is the number of inputs to the chip. For example, a 100-circuit chip with 50 inputs would mean 10^{15} tests, with many more possible if the chip has internal storage states or the sequence of testing must be varied. Fortunately, most of these tests are redundant, at least to some extent, or may otherwise be compromised. However, it is still a major problem to

define and develop an efficient and properly qualified set of tests. In principle, we may have a solution for the d.c. functional test problem,[15,16] even for sequential logic, but there has to be a serious question about the economics of the situation for the limited volume machine.

There are sophisticated test generation programs in operation today which were developed to test cards of conventional logic. Although these techniques are extendable to LSI, they were developed in a less stringent environment and impose constraints, as they presently exist, which limit our flexibility in LSI. For example, there are restrictions in the number of levels of sequential logic and if this is observed, the logic on the chip would have to be broken up at the expense of I/O pins and performance. Several companies have reported work here but it is not clear that adequate general-purpose test programs have been completed yet.

Simulation

Simulation of the potential LSI machine or components, either by software or unit logic hardware or some combination of both, is the accepted means of detecting and correcting design errors before production. However, there is at present no completely effective and comprehensive simulation system to detect all system errors, although there are techniques available aimed at most aspects of system simulation.[17,18]

There are a number of programs for higher level system simulation that can be applied down to a rather detailed level. Such programs can simulate the operation of a system quite accurately, but the problem of detecting design errors also requires that the system users as well as designers describe completely all the ways in which they expect to use the system. This is one of the main problems in system simulation. A system has many parts that are expected to operate concurrently, and the design should contain interlocks to prevent improper sequencing of operations that use several semi-autonomous machine parts. The designer cannot always anticipate all the combinations of applications of the machine parts or sequences that a user may employ. Sometimes the designer incorrectly assumes that certain combinations either don't make sense or result in don't-care conditions. This problem is compounded somewhat by the fact that a software system may be expected to take care of some undesirable conditions without full knowledge of the time and memory needed by the software system. Simulation of the architecture of a machine system is intended to encompass the hardware and software operation of the system but even in this level, the architect cannot foresee, in a complex system, all the anticipated combinations of conditions dependent upon both the programs and data that may occur in some practical application. To investigate exhaustively all possible system states or conditions in detail, even for fairly simple systems, is out of the question because of the years of time that would be required.

The only reasonable approach to large-system simulation is to simulate at several different levels of detail, and at each level to simulate a limited number of units, each with limited complexity. At the higher levels of complete system simulation, one is primarily trying to check out broad timing properties of the system and insure that improper sequences of events do not occur. Simulation at an intermediate level will determine whether the functional units of the system will individually process data correctly. At the most detailed level of simulation, real circuit-to-circuit delays and the basic system synchronization techniques become important. If the system is basically asynchronous then the logical interlocks need to be carefully checked; but not detailed circuit delays. If the system is not basically asynchronous detailed wiring distance and circuit loading delays become important. As far as LSI is concerned, one problem is to insure that all timing paths on semiconductor chips stay within certain tolerances. If the design tolerances are too loose, the system will operate too slowly. If the tolerances are too tight, the yield of acceptable chips may be too low.

An alternative to software simulation in the past has been hardware simulation. This has usually taken the form of a prototype model built with pre-production circuits. For LSI, this hardware simulation may be done with unit logic circuits provided they can be designed and connected to realistically simulate LSI circuits and chip characteristics. This is still the path contemplated by many users.

SUMMARY

LSI use will be very much a function of component or system quantities, and even more, there will be a significant proliferation of logic circuitry within systems having large market quantities, e.g., terminals. The market volumes necessary for profitable LSI use will decrease as more economical solutions evolve but the "few-of-a-kind" components may never be justified by cost considerations alone.

The main concern of users in general is turn-around time (and costs) in the face of the many design changes they anticipate. Highly automated schemes and more effective communication between the user and the component manufacturer obviously hold the key. Excellent progress is being made in many areas, particularly in mask-making, partitioning and testing, but much remains to be done.

ACKNOWLEDGMENT

We would like to acknowledge support from P. Cook, D. Critchlow, H. Freitag, E. Schischa, J. L. Smith, and S. Triebwasser.

REFERENCES

1 R C MINNICK
Survey of microcellular research
Journal of the ACM April 1967

2 D L SLOTNICK
Achieving large computing capabilities through an array computer
AFIPS Conference Proc vol 30 April 1967

3 J GREGORY R McREYNOLDS
The SOLOMON computer
IEEE Trans on Electronic Computers vol EC-12 no 6 December 1963

4 J HOLLAND
Iterative circuit computers
Proc WJCC 1960

5 W COMFORT
Highly parallel machines
Proc of 1962 Workshop on Computer Organization Spartan Books Washington D.C. 1963

6 W NOTZ E SCHISCHA J L SMITH M G SMITH
LSI—The effect on systems design
Electronics February 20 1967

7 S TRIEBWASSER
Programmable interconnection techniques
ISSCC February 11 1966

8 H FREITAG
Design automation for large-scale integration
WESCON Los Angeles August 25 1966

9 J S KILBY
Device fabrication
ISSCC February 9 1966

10 M CANNING R DUNN G JEANSONNE
Active memory calls for discretion
Electronics February 20 1967

11 C H MAYS
Computer-aided design for large-scale integration
ISSCC February 16 1967

12 N L HAZLETT
Computer accelerates design and production of large arrays
Electronics February 20 1967

13 D L CRITCHLOW
Layout and mask generation for large-scale integration
IEEE international convention March 23 1967

14 P W COOK G A LEMKE A BRENNEMANN
An automatic integrated circuit mask artwork generating system
Microelectronics Symp St Louis Mo June 20 1967

15 E EICHELBERGER
Hazard detection in combinational and sequential switching circuits
IBM Journal March 1965

16 J ROTH W BOURICIUS P SCHNEIDER
Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits
Available from IEEE Computer Group Repository—received Dec 9 1966

17 D TEICHROEW J LUBIN
Computer simulation—discussion of the technique and comparison of languages
Communications of the ACM October 1966

18 R LARSEN M MANO
Modeling and simulation of digital networks
Communications of the ACM May 1965

A family of linear integrated circuits for data systems

by MARVIN B. RUDIN, RICHARD L. O'DAY
and R. H. JENKINS
Fairchild Semiconductor
Palo Alto, California

INTRODUCTION

The rapid expansion of automated digital data processing has created a need for low-cost analog data acquisition, display and control equipment. This implies low-cost D/A and A/D conversion equipment. This paper describes the resulting family of linear integrated circuits satisfying this need.

Conversion subsystems requirements

The principal parameters affecting the broad utilization of data converters are speed, accuracy and cost. Traditionally, higher word rates and accuracy are demanded of analog to digital converters than that of digital to analog. This is because: (1) multiple measurements are time multiplexed for encoding by a single A/D, whereas D/A commonly are used singularly; (2) A/A measurement range is employed wastefully for greater input flexibility, whereas analog output range tends to be standardized; (3) analog displays are limited in accuracy and speed by the resolving power and frequency response of the equipment as well as the resolution capability and flicker response of the eye; (4) high speed transient phenomena have to be measured in real time but are generally presented in post time; (5) A/D's generally operate bit-sequentially compared to D/A's operating bit-parallel.

A survey of data acquisition, display and control systems has shown an acceptable specification compromise between volume converter usage and mass production capability to be:

D/A Conversion

Rate	DC to 50K words/sec.
Resolution	8 to 10 Bits
Accuracy	±0.2% (full scale) ±0.1% (non-linearity)
Analog Range	0 to +5V or 0 to +10V
Output Current	20ma
Reference Voltage	Internal
Logic Compatibility	Bipolar (Micrologic families of CCSL, μL, RTμL, MWμL, DTμL, TTμL, and CTL).
Code Format	Binary or BCD
Temperature Range	O°C—70°C (Industrial) −55°C to +120°C (Aerospace)

A/D Conversion

Put-through rate	DC to 100K words/sec
Resolution	10 Bits
Accuracy	±0.05% (non linearity) ±0.001%/day (fullscale drift @ +25°C) ±0.001%/°C (fullscale drift with temp)
Analog Range	0 to 5V or 0 to +10V.
Analog Input Impedance	>1 megohm
Reference Voltage	Internal with external trim optional
Logic Compatibility	Bipolar (Micro logic families of CCSL, μL, RTμL, MWμL, D7μL, T7μL, and CTL)
Code Format	Binary or BCD
Temperature Range	0—70°C (Industrial) −55°C to +125°C (Aerospace)

Selection of a conversion technique

The goal of the subsystem design was to satisfy the stated specifications at a minimum cost to both the manufacturer and user. Therefore the design economics

must be considered first in the selection of a conversion scheme. The prerequisites for low cost for the manufacturer are:

1. Batch fabrication
2. High volume
   a. Minimum number of different chip types
   b. Each component independent functionally and individually saleable
3. High processing yield
4. Minimum amount of hand assembly work such as lead bonding.
5. Capable of simple or automated testing.

The prerequisites for low cost to the user are:

1. Easy to understand and use
2. Minimum number of external components and connections for assembly and test
3. Minimum number of device types to procure and stock
4. Compatible with existing hardware
5. Flexible in design so may be adaptable to many different applciations within the user's system(s) and product lines.

The factors thus stated imply a system which utilizes integrated circuits to the maximum extent permitted by the existing semiconductor technology. Monolithic construction provides the means for low cost production and a large number of functions per package, thus minimizing interconnections while enhancing system reliability.

The subsystem specifications of speed, temperature performance, and long-term stability could only be satisifed by bipolar transistor design for both the linear and digital portions of the circuit. Also, the logic levels and power supply voltages required for other than bipolar design would not be compatible with the majority of existing data equipment.

Preparatory to circuit design the known ADC and DAC configurations were studied.[1] Those techniques that would satisfy the subsystem performance specification and allow a major portion of the functions to be monolithically integrated were cataloged for component type and approximate count. A summary is tabulated in Figure 1.

The choice of a D/A technique was limited to either switched voltages/currents or switched resistors. Current switching was chosen because: (1) it is independent of switch off-set voltages; (2) it is fast and may be accomplished with a minimum of transient problems; (3) control circuit isolation is easily accomplished; (4) mutual isolation of current sources and output is easily accomplished.

The A/D techniques ranging from the ramp counter to cyclic, successive approximation, multi-comparator, ripple-through and parallel types were considered.[2,3] The choice was simplified by the following: the ramp-counter types are too slow; multi-comparator, ripple through and parallel types require excessive circuitry; and the cyclic types require accurate voltage switching in conjunction with large external capacitors. A D/A feedback successive approximation converter easily satisfied the performance specifications and could use the D/A as a feedback element. It offered these salient advantages:

Type of Converter / Required Components	D/A Techniques		A/D Techniques				
	R-2R Resistive Ladder	Weighted Voltages or Currents (Binary BCD Octal)	Cyclic (X2)	Cyclic (Charge Division)	D/A Feedback (R-2R)	D/A Feed-back (weighted) VORI)	Cascade X2
Analog Switches	20	10	7	10	10	10	20
Precision Resistor	21	10	0	0	21	10	0
Precision Capacitors	0	0	0	3	0	0	0
" Gain Amps(fast)	0	0	2	1	0	0	10
" Gain Amps(slow)	1	1	1	1	1	1	1
" Ref Source	1	1	1	1	1	1	1
" Comparators	0	0	2	1	2	1	10
Large STORAGE Capacitors	0	0	2	3	0	0	0

Figure 1—Required component comparison for 10 bit converters

1. Generally well-known and understood and thus easy for the user to become familiar with, and use.
2. Versatile as the same building blocks may be employed for both A/D and D/A operation.
3. Coding flexibility—applicable to any D/A code.
4. Adaptable to all IC functional blocks for ultimate economy. The commonality of chip types for A/D and D/A means lower development and manufacturing costs.
5. The functional blocks are independently saleable products.

Because of the above considerations the design centered around a bipolar current summing D/A converter. The problem became one of defining the subsystem breakdown on a circuit building block basis, relative to IC technology capabilities.

DA converter configuration

Functional blocks needed for the DA subsystem, as shown in Figure 2 include:[4]

1. Transimpedance Amplifier
2. Logic buffer/current switching
3. Current sources
4. Voltage reference
5. Data register

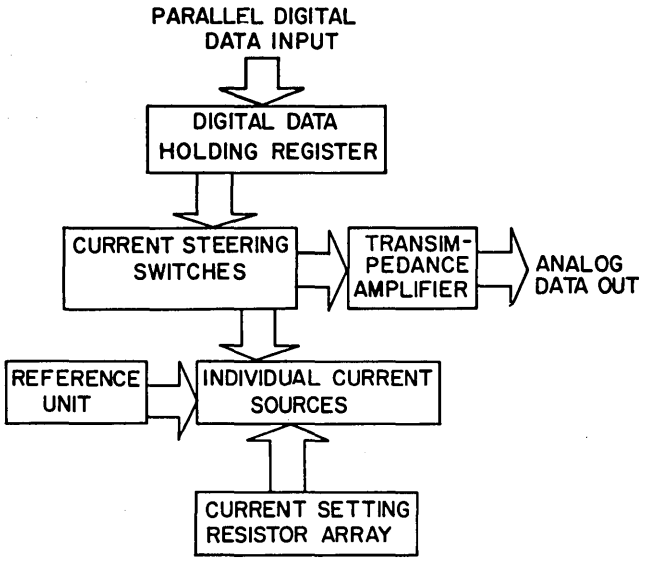


Figure 2—Current summing digital to analog converter functional blocks

Adequate integrated summing amplifiers are currently available in large quantities. Because these amplifiers find application in other areas their production volume is already high and their price, therefore, attractive. Further, such a choice allows the user to make the cost/performance compromises and tailor the converter to his specific system requirements.

A simplified version of the current switching circuit is shown in Figure 3A. It is apparent that current will flow through the diode whose anode is at the higher potential. By maintaining the output of the current switch at a fixed potential the current flow can be determined by applying various voltages to the "control" terminal. One disadvantage of this circuit is that the control potential must be capable of supplying all of the switched current. A significant improvement on this circuit is realized by driving the control from an emitter follower as shown in Figure 3B. The addition of a current source level shifts the control potential low enough in the "ON" condition to insure full current flow in the output. Summing any number of switched currents takes place by merely connecting the outputs of two or more current switches in parallel. However, care must be exercized to insure that the reverse diode leakages are small to minimize error current in the output. Quality silicon fabrication gives adequately low leakage over the temperature range for ten switches in parallel. Even though the switches are not gold doped, recovery is rapid enough for the required megabit operation since only one diode per switch is saturated.

To complete the explanation of the D/A converter

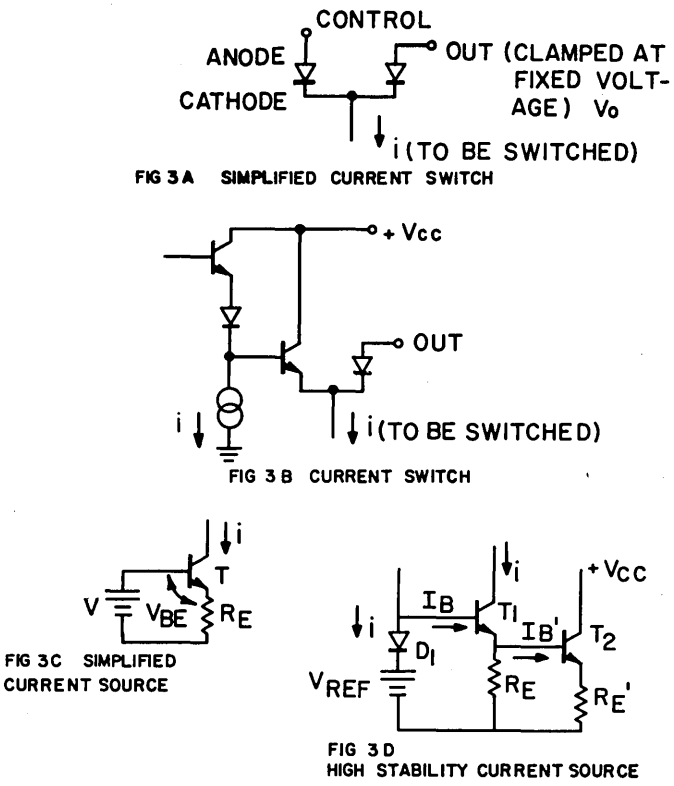


Figure 3

scheme, only the generation of accurate and stable currents remains. Typically in bipolar circuitry, current sources are made using the basic circuit shown in Figure 3C.

The collector current of such a circuit is given by $i_c = \alpha\frac{V-V_{BE}}{R_E}$. For most transistors, α approaches unity and V_{BE} is a logarithmic or weak function of collector current. As a result a reasonable current source is obtained. Because the output conductance of most transitors is low, the collector voltage of the current source has little effect on the collector current. For DA conversion where high accuracy and stability of the current value is required, steps must be taken to eliminate the collector current dependence of α, and thermal variations of V_{BE}. In Figure 3D a more complex current source is shown which remedies these variations.

The circuit is designed so that $I_B = I_B'$, therefore forcing the collector current equal to the current through R_B, the current determining resistor. This eliminates the dependence of the collector current I_c on the transistor current gain, α. Because T_1 and T_2 are on the same chip only a few mils apart, their characteristics are closely matched. In addition the use of high gain transistors minimizes $|I_B - I_B'|$.

Because a 10 bit converter requires 10 current

sources whose currents must be related in a binary manner, the V_{BE} dependence of I_C must be accounted for. Fortunately the collector current density is given by $J_c = J_{(sat)} \exp\frac{qV_{BE}}{kT}$ for many orders of current magnitude and over the military temperature range.[5] Thus, by knowing the design current value, a given V_{BE} may be determined and compensated if the emitter areas are held such that

$$\frac{I_c}{A} = \frac{I_{sat}}{A} \exp\frac{q\,V_{BE}}{k\,T}.$$

Much experience has been obtained in the matching of transistor parameters through the production of differential operational amplifiers. This experience indicates that the fabrication implications of such a requirement are not severe. Since D_1 has similar geometry to T_1 and its current matches T_1, the temperature dependence of V_{BE} may be compensated. Thus, by choosing appropriate values of R_E, stable current generators are available. The technique of combining this type of current source with current switching makes switching speed independent of mode capacitances in the current setting resistors, because the current continuously flows through the precision resistors.

None of the preceding considerations precludes the combination of the switches and the current sources on the same chip. Indeed, the use of high gain transistors improves the quality of the current sources, while the longer lifetime material required by these transistors decreases the leakage currents in the switches. However, a compromise must be made in the minority carrier lifetime of the material, for if it is too long, switching speed will degrade.

It is also apparent that the value of the current sources will depend on the value of the reference voltage. Because a zero temperature coefficient reference can also be fabricated using the high gain process, this too was included in the chip. Provision was made for the user to supply his own external reference for either (1) greater stability or, if required, (2) analog multiplication.

The digital register was not included on the current source chip for the following reasons.

1. Marginal speed without gold doping.
2. Incompatability with high gain used in the current sources.
3. Chip size would be inordinate for quantity production with present state of the art.
4. Avoidance of possible redundancy with respect to digital system registers.

With very few compromises it is possible to integrate the current sources, the current switches and the reference on the same chip at no loss of versatility for the user and considerable gain in ease of use. By leaving the precision resistors off the chip the user may use whatever codes he desires, thus enhancing flexibility.

With the inclusion of these three functions on the silicon chip, an area 60 by 160 mils was required. By using a proven process and designing with non-critical masking tolerances, the best possible yields were assured for this large circuit. In addition, the circuit utilizes only NPN transistors, ten of which require matching equivalent to integrated differential amplifier input transistors. The proven process consists of the standard 6 mask, monolithic epitaxial integration typical of currently available LIC's. Figure 4 shows the circuit.

In the final D/A configuration (Figure 5) the integrated blocks consist of the data register, the summing amplifier, the binary weighted current sources with the switches and reference, while the resistors are separate. The use of these functionally independent blocks allows the system designer to meet his conversion requirements at minimum cost.

The D/A performance curves for full scale drift and non-linearity versus temperature are shown in Figures 6 and 7.

A/D converter configuration

As mentioned earlier the DAC is utilized as a feedback element for a high speed successive approximation analog to digital converter (see Figure 8). The general comments made for the D/A converter apply also for the A/D converter. Operation of the A/D configuration in Figure 8 is as follows:

The logic programmer will successively try each data bit starting with the most significant (MSB). The programmer will monitor the comparator output to determine if the bit value is too large or too small. If not ($I_sR_s \leqq V_x$), the comparator will cause the digital data register to hold the bit in. If the bit value pulls the summing bar negative ($I_sR_s > V_x$), the comparator will cause the logic to remove the bit. The programmer then will try the next bit in succession until $V_s \rightarrow 0$ and the digital equivalent of the analog signal (V_s) is stored in the register.

The summing current levels of the DAC, for 10-bit operation, are not directly compatible with the temperature-dependent offset current of most presently available IC comparators (e.g., μA710). A thermally stabilized differential pair μA726 may be used as an excellent buffer stage. For moderate temperatures, a simple differential pair is satisfactory. IC comparators of the μA710 class may be used directly for high speed, low accuracy operation (i.e., 6 to 7 bits) over a limited temperature range.

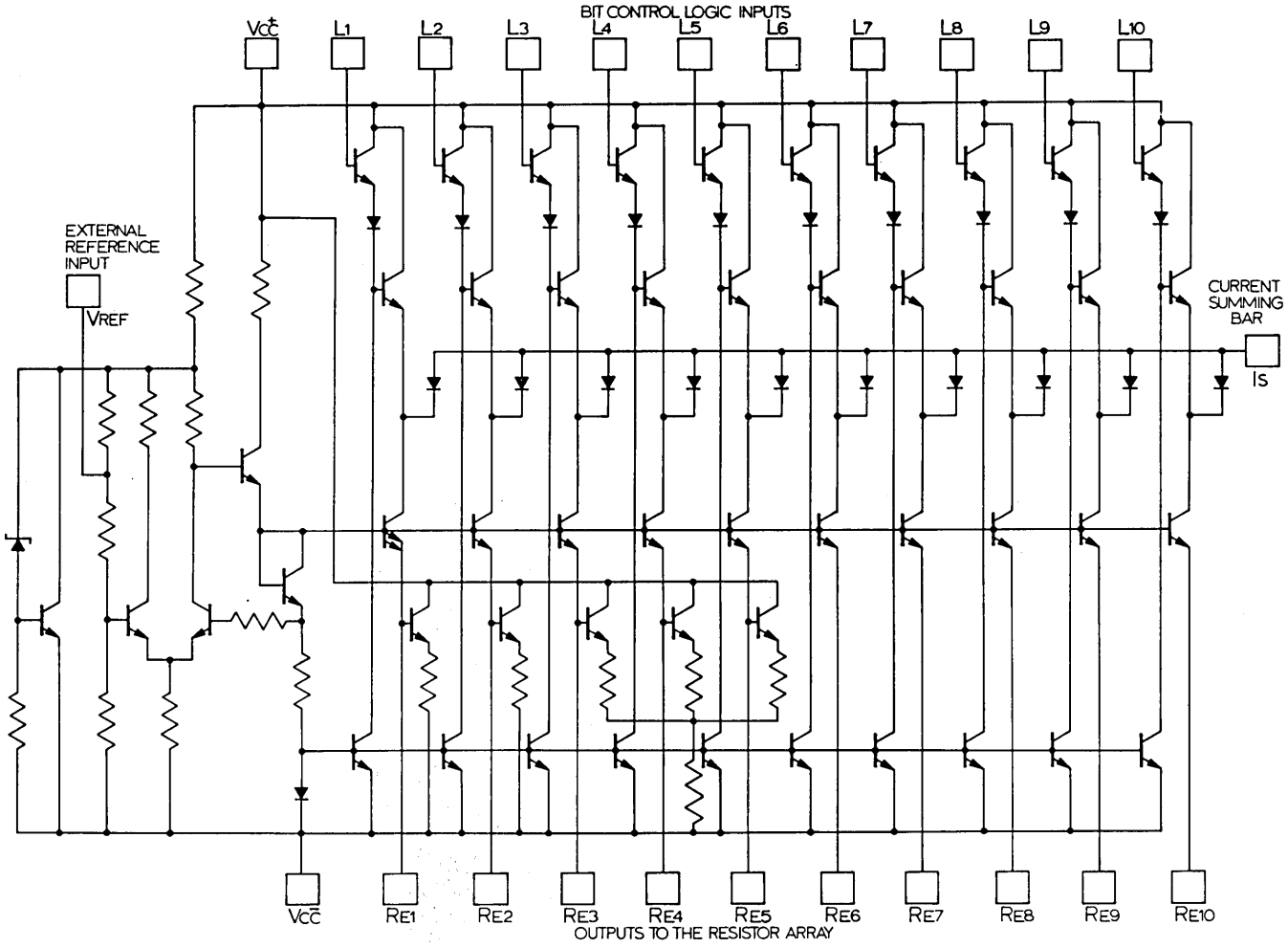


Figure 4—I.C. digital to analog converter

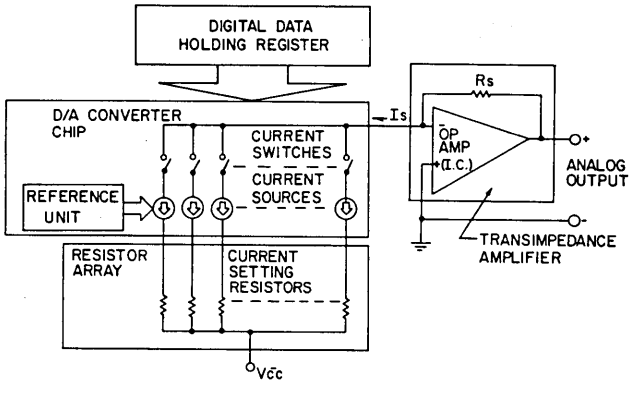


Figure 5—Current summing D/A converter I.C. blocks

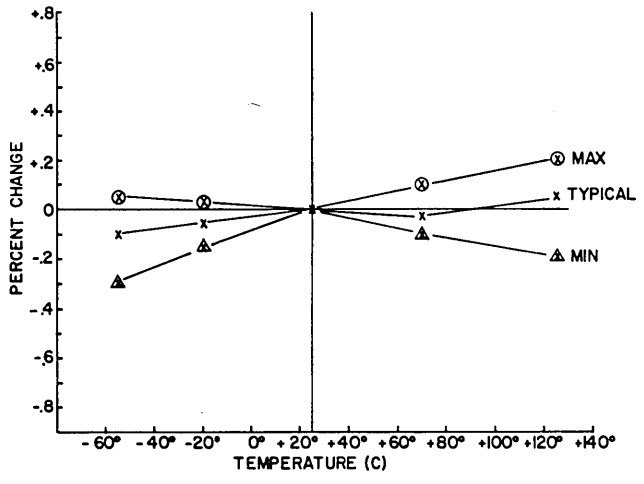


Figure 6—Percent change in full scale current vs temperature

Another solution is the use of a high slewing rate operational amplifier driving a comparator such as the μA710 in the place of the IC comparator. At present there is a high slewing rate (30V/μsec) operational

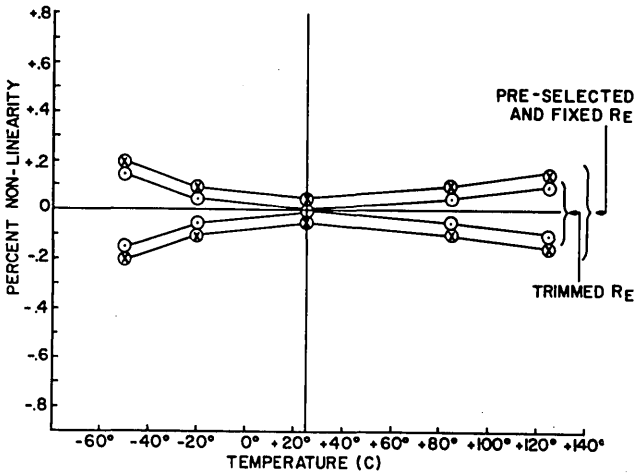


Figure 7—Percent non-linearity vs temperature

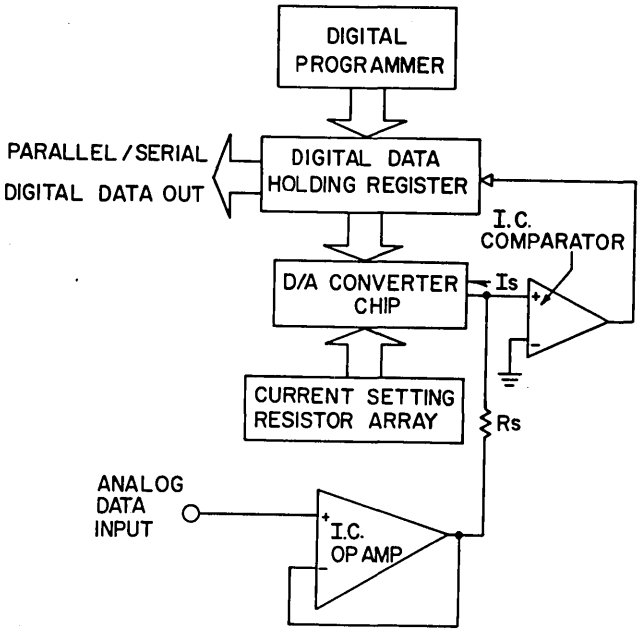


Figure 8—Successive approximation current summing A/D converter

amplifier in development capable of operation in this configuration.

Eventually a precision IC comparator (in development) will be available which is directly compatible with the D/A current sources and requires no buffering. Performance of the new comparator amplifier is summarized as follows:

V_{offset}	OmV (externally trimmed)
$V_{offset(T)}$	$5\mu V/°C$
I_{offset}	25nA
$I_{offset(T)}$	1.6nA/°C
$\tau_{switching}$	300 nsec.
V_{in} range	±5V
Common mode rejection	90dB
Power supply range	±6V to ±15V
Power supply rejection	$100\mu V/v$

Its outstanding input characteristics are made possible by a new IC process which provides substantially reduced offset current and current noise, in conjunction with higher transistor gain than present linear IC processes.

The new comparator will permit up to 13 bit resolution and similar accuracy. With nominal temperature stabilization 15 bits can be achieved.

Performance of the IC converters

A typical D/A converter is shown by Figure 9. The performance that may be expected is:

I.C. D/A Performance Parameters

Rate	DC to 50K words/sec.
Resolution	10 bits (Binary)
Accuracy	±0.2% (full scale) ±0.05% (non linearity)
Analog range	Variable 0 to +12 volts (Max)
Output current	20ma.
Reference voltage	Internal with optional external trim
Logic control levels	"1" less than +0.5VDC "0" greater than +2.5VDC
Code format	10 bit binary 8 bit BCD
Temperature range	−20°C to +125°C (Specification) −55°C to +125°C (Operating)

An A/D configuration is shown by Figure 10. The performance parameters are:

I.C. A/D Performance Parameters

Put-through rate	DC to 50K words/sec (10 Bit Binary Accuracy)
Resolution	10 bits (Binary)
Accuracy	±0.05% F.S. (non-linearity) ±0.01% F.S/DAY (fullscale drift @ 25°C) ±0.005% F.S/°C (fullscale drift with temp)

Analog range	Variable, 0 to +12 volts (max.)
Analog input impedance	>1 megohm
Reference voltage	Internal with external trim optional
Logic Control Levels	"1" less than +0.5VDC "0" greater than +2.5VDC
Code format	10 Bit Binary 8 Bit BCD
Temperature range	−20°C to +125°C (Specification) −55°C to +125°C (Operating)

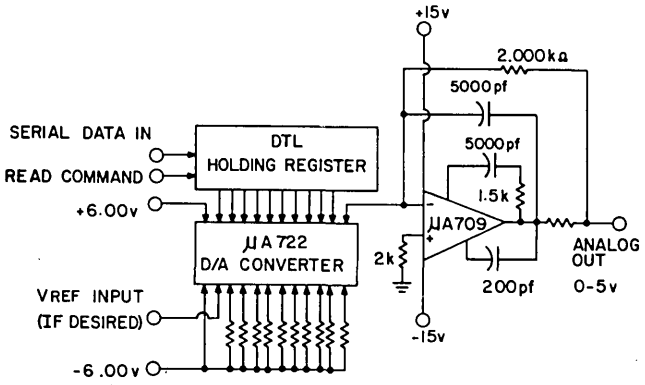


Figure 9—D/A converter

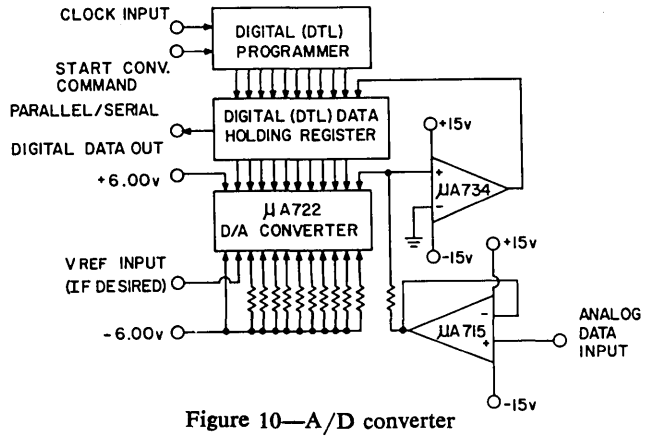


Figure 10—A/D converter

The resistor arrays used in two applications were discrete metal wire-wound devices. Film resistor (thin or thick) may be used, as the array values may be pre-selected to achieve the accuracies stated. If trimmed arrays to match the current sources are desired, the non-linearity error can be reduced to zero at +25°C. Also the components are small enough to easily fit within a P.C. board-mounted proportional control oven. These would allow paralleling units for greater accuracies and 13-15 bit resolution.

CONCLUSION

New IC functional blocks permitting all I.C. analog digital data converters are now nearing production. As with I.C. logic elements, the cost to the user can be expected to fall to the point where economies will grossly change design philosophies in the data acquisition field. These do-it-yourself components will make low-cost analog/digital peripheral subsystems a true reality.

REFERENCES

1 A K SUSSKIND
Notes on analog-digital conversion techniques
The Technology Press MIT and John Wiley & Son Inc Chap 5 1957

2 K HINRICHS
Digital to analog conversion equipments and techniques
1964 Systems Engineering Conference New York N.Y. June 1964

3 B D SMITH
Coding by feedback methods
Proc IRE vol 41 no 2 pp 1053-1058 August 1953

4 M B RUDIN R L O'DAY R T JENKINS
System circuit device considerations in the design and development of a D/A and A/D integrated circuits family
1967 International Solid State Circuits Conference University of Pennsylvania February 1967

5 C T SAH
Effect of surface recombination and channels on p-n junction ana transistor characteristics
IRE Transactions on Electron Devices pp 94-103 January 1962

The effect of digital compensation for computation delay in a hybrid loop

by EWART EDWARD LESLIE MITCHELL
Electronic Associates Inc.
Princeton, New Jersey

INTRODUCTION

Recent interest in hybrid computation has focussed attention on the errors incurred because of the method of solving the problem equations. In the past some attention had been given to errors incurred due to analog component accuracy or round off error in a digital solution. However, with the advent of hybrid computation and the knowledge that the sampling that had to occur at the interface would perturb the solution, a number of papers have been published to define the problem and propose solutions. Probably the reason for the stimulated interest is that the errors can be obtained analytically for a number of simple systems. The proposed solutions to the sampling delay have shown an order of magnitude increase in the accuracy of the selected problems to which they were adapted.

It is the purpose of the paper, however, to show that in general it is *not* possible to compensate for computation delay with a digital filter unless the basic damping of the system under study remains relatively constant over the duration of the problem or the sample interval can be held to 30 or more per cycle.

Miura and Iwata[1] considered the effect of digital execution time and showed that the delay would influence the poles of the system under study, obtaining expressions for the amount of the shift for a number of simple systems. To compensate for the digital execution time they suggested three compensation techniques:

(a) Assuming that the digital output is immediately integrated, the output of the integrator is modified by adding $(\tau+T/2)$ of the input. Here τ is the delay due to the digital computation and T is the hold time at the output, where normally $T = \tau$. See Figures 1 and 2.

(b) The digital computer can predict ahead by linear interpolation, i.e.,

$$y = y'_j + (\tau + T/2)\frac{(y'_j - y'_{j-1})}{T}$$

(c) The analog first order hold can be implemented in order to avoid the staircase output and provide the necessary lead. Here y'_j is the computed output of the digital computation at the j^{th} time step; y_j is the output compensated for the delay.

Figure 1—Definition of time T and τ

Figure 2—Compensation by modified integration (Miura & Iwata)

Of these techniques the first one is the only one that doesn't increase the order of the system. The other two introduce extra roots into the system equations by

using past history, which can be detrimental to solution accuracy under certain conditions.

Karplus[2] has also examined the effect of digital execution time on the solution error. In addition to the compensation techniques specified above, he also suggets modifying the *input* to the digital computer—if it is the result of an integration—by $(\tau + T/2)$ times the derivative. In a sense this is equivalent to (a) above and does not affect the order of the system.

Gilbert[3] also suggests modification of the input to the digital computer when the derivatives are known in a similar manner as above. Modifying the input to the digital section avoids discontinuities in the outputs of the analog integrators. Jumps occur when the compensation of Miura and Iwata is used.

The evaluation suggested by both Karplus and Gilbert for the compensation technique is accomplished by observing the growth or decay of a simple sine wave oscillator, which is very sensitive to phase shift. Unfortunately, these compensation techniques work very well for lightly damped systems; it is the heavily damped systems that produce the big errors.

Matlock[4] extends the digital prediction technique to higher order filters to better compensate for the phase shift. These higher order filters introduce extra roots into the system which seriously affect the performance of *heavily* damped systems. This digital prediction technique has also been utilized by Deiter and Nomura[5] who obtain the prediction coefficients by Gregory-Newton extrapolation. They evaluate the technique by measuring the integral error in the representation of a sine wave oscillator. Again the choice of a lightly damped system for evaluation leads to false confidence in the technique.

Our experience with predictive filters came about a year ago when we applied the scheme to a hybrid helicopter simulation. The forces and moments are computed on the digital section and integrated on the analog. To stabilize the short period loops we included 2nd order (quadratic) prediction in the moment equations. The first evaluations—at hover—indicated that the match with the real world was extremely close. Unfortunately, when we approached the top speed, the tail started to wag, at a frequency about eight times higher than expected, and the system went unstable. When the prediction subroutine was removed, the instability was also eliminated and left us with a simulation that had frequency and damping still reasonably close to the actual vehicle.

The explanation lies in the extra roots introduced by the prediction technique and the fact that the natural air frame is heavily damped at high speed. We will show that the application of prediction to a heavily damped system can lead to gross errors in the simulation when the extra roots move into the region of interest—near the unit circle in the z-plane or the $j\omega$ axis in the s-plane.

To overcome the difficulty we had to adopt a technique similar to that proposed by Gelman[6] or Connelly.[7] In essence a simple analog model is built up to approximate the system as closely as possible, and the digital computer is used to determine errors in the analog model. Provided these errors remain small correction terms, then the major feedback loops are continuous and adequate simulation accuracy can be maintained. Adequate compensation should also have been obtained using the input compensation of Karplus or the first technique (a) suggested by Miura and Iwata, since the order of the system is not modified.

In order to demonstrate the difficulties inherent in digital prediction, consider the simulation of a simple second order system and let us examine the effects of different predictive filter coefficients and damping factors on the roots of the system equations. Naturally, actual systems will be of higher order and contain nonlinearities, but significant trends can be unearthed by treating the simplest possible configuration.

Problem statement

In order to represent a physically meaningful system let us consider the simulation of vehicle pitch plane dynamics, where the velocity vector is held constant. The pitching acceleration, M, is assumed a function of pitch rate, Q, and angle of attack, α. The differential equations to simulate the motion are:

$$\dot{Q} = M$$

$$\dot{\alpha} = Q$$

Note that we have absorbed the moment of inertia into the moment M, so that M has the dimensions rad/sec^2 rather than the conventional ft-lbs.

Our digital computer is programmed to compute M, and we will assume that this is linearly related to the problem state variables. In actual practice, the computation of the forces and moments requires almost all the available digital computation time; but for the purposes of analysis we will consider perturbations about a steady state.

Expressing

$$M = M_q Q + M_\alpha \alpha$$

the state equations become

$$\begin{pmatrix} \dot{Q} \\ \dot{\alpha} \end{pmatrix} = \begin{pmatrix} M_q & M_\alpha \\ 1 & 0 \end{pmatrix} \begin{pmatrix} Q \\ \alpha \end{pmatrix}$$

The frequency and damping for the unforced continuous system are obtained from the eigenvalues of the

state matrix or roots of:

$$\lambda^2 - M_Q\lambda - M_\alpha = 0$$

so that

$$2\zeta\omega_n = -M_Q$$

$$\omega_n^2 = M_\alpha$$

where ζ is the damping and ω_n the natural frequency of the undamped system.

In the s-plane the roots are at $M_Q/2; \pm [M_Q^2/4 + M_\alpha]^{1/2}$ and it will be our objective to determine how close the roots of the compensated system approach these desired values.

Now to compensate for the sampling effects at the analog-digital interface we will apply up to quadratic prediction to the computed M by taking a linear combination of the current value and past history—up to 3 time steps ago. See Figure 3.

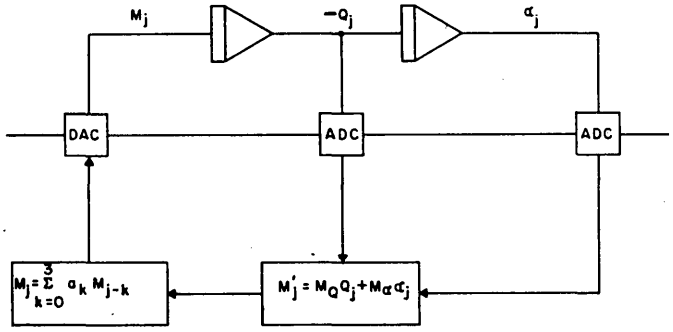


Figure 3—Flow diagram for hybrid damped sine-wave oscillator

i.e.,

$$M_j = \sum_{k=0}^{3} a_k M'_{j-k}$$

where

$$M'_j = M_Q Q_j + M_\alpha \alpha_j$$

and j is the time step count, incrementing in the sample time, T.

We want to consider the effect of various values of the predictor parameters $a_0 - a_3$ on the solution. The following interpretation can be made for the a's:

(a) $a_0 = 1$

$a_1 = a_2 = a_3 = 0$

The system corresponds to the case where the digital computation time is zero, the error being introduced by the necessity to hold the output constant over the sampling time, T.

(b) $a_0 = 0$

$a_1 = 1$

$a_2 = a_3 = 0$

This condition is observed in most current hybrid simulations where the computation time is equal to the sample time. Thus, the output, M, is delayed from the two inputs by one complete cycle.

Other coefficients may be obtained by solution of the set of simultaneous equations given in the appendix. The essential requirement of the prediction is the attempt to recover the lost 1.5 time steps ($=\alpha$) due to the combination of digital computation time and sample hold.

From Figure 3, the basic state equation can be expanded to develop the transition matrix that transfers the state from one time step to the next.

$$Q_{j+1} = Q_j + \int_0^T M_j dt = Q_j + M_j T$$

$$\alpha_{j+1} = \alpha_j + \int_0^T Q\, dt = \alpha_j + Q_j T + \tfrac{1}{2} M_j T^2$$

Substituting for M_j, where

$$M_j = \sum_{k=0}^{3} a_k M'_{j-k}$$

&

$$M'_j = M_Q Q_j + M_\alpha \alpha_j$$

leads to:

$$\begin{bmatrix} Q_{j+1} \\ Q_j \\ Q_{j-1} \\ Q_{j-2} \\ \alpha_{j+1} \\ \alpha_j \\ \alpha_{j-1} \\ \alpha_{j-2} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \end{bmatrix} \begin{bmatrix} Q_j \\ Q_{j-1} \\ Q_{j-2} \\ Q_{j-3} \\ \alpha_j \\ \alpha_{j-1} \\ \alpha_{j-2} \\ \alpha_{j-3} \end{bmatrix}$$

where

$a_{11} = 1 + a_0 M_\alpha T$ $\quad$ $a_{41} = T + a_0 M_Q T^2/2$

$a_{12} = a_1 M_Q T$ $\quad$ $a_{42} = a_1 M_Q T^2/2$

$a_{13} = a_2 M_Q T$ $\quad$ $a_{43} = a_2 M_Q T^2/2$

$a_{14} = a_3 M_Q T$ $\quad$ $a_{44} = a_3 M_Q T^2/2$

$a_{15} = a_0 M_\alpha T$ $\quad$ $a_{45} = 1 + a_0 M_\alpha T^2/2$

$a_{16} = a_1 M_\alpha T$ $\quad$ $a_{46} = a_1 M_\alpha T^2/2$

$a_{17} = a_2 M_\alpha T$ $\quad$ $a_{47} = a_2 M_\alpha T^2/2$

$a_{18} = a_3 M_\alpha T$ $\quad$ $a_{48} = a_3 M_\alpha T^2/2$

Now we can see that by adding in the prediction—or compensation—terms, we have increased the order of the system under study from 2nd to 8th, at first glance, for the quadratic predictor. However, the rank of the matrix is only 5, so just three extra roots are introduced. The eigenvalues of this transfer matrix correspond to poles in the z-plane for the sampled system. It is immediately obvious that for stability, all the eigenvalues must lie within the unit circle, otherwise successive application of the matrix will cause (some) solutions to increase without bound.

This eigenvalue equation can be reduced to:

$$[1 + a_0M_qT - a_0M_\alpha T^2/-(2 + a_0M_\alpha T^2/2)\lambda + \lambda^2]\lambda^6 + T[(1-\lambda)M_q - (1+\lambda)M_\alpha T^2/2]\,[a_1\lambda^2+a_2\lambda + a_3]\lambda^3 = 0$$

Three of the eigenvalues are zero, but we are left with a fifth order polynomial or three extra roots. The characteristic polynomial can be rearranged to:

$$\lambda^5 + b_1\lambda^4 + b_2\lambda^3 + b_3\lambda^2 + b_4\lambda + b_5 = 0$$

where

$$\begin{aligned} b_1 &= -a_0\gamma - 2 \\ b_2 &= -a_1\gamma + a_0\zeta + 1 \\ b_3 &= -a_2\gamma + a_1\zeta \\ b_4 &= -a_3\gamma + a_2\zeta \\ b_5 &= a_3\zeta \end{aligned}$$

and

$$\begin{aligned} \gamma &= T(M_q+M_\alpha T/2) \\ \xi &= T(M_q-M_\alpha T/2) \end{aligned}$$

We can show that for $T << 1$, the three extra roots move into the origin as well, leaving the two that correspond to the original system near the $+1$ point.

Transforming these to the s-plane by:

$$s = \frac{1}{T}\log_e z$$

results in roots at:

$$s = M_q/2 \pm [M_\alpha + M_q^2/4]^{1/2} + 0(T)$$

so that we have a check that the sampled system corresponds to the original continuous system for small sample times.

Our object now is to study the behavior of the root loci obtained by varying T for different prediction parameters $a_0 - a_3$ and for varying damping M_q. We have kept $M_\alpha = 1$ corresponding to a natural frequency of $1/2\pi$ cycles/sec; and on all the root loci, sample time, T, is expressed as samples/cycle.

Results

Four cases of practical interest were examined and applied to systems that had inherent high and low damping. It would appear from this that at low sampling rates, the extra roots introduced by the prediction can cross the imaginary axis and become unstable, especially at high damping ($M_q = 1.0$).

Case I: Figure 4 shows the root locus for the system,

$$\begin{aligned} a_0 &= 1 \\ a_1 = a_2 &= a_3 = 0 \end{aligned}$$

for the two values of damping $M_q = 0.2$ and 1.0. These loci correspond to the unrealized case of zero digital computation time; but since it is the only one where only two roots are maintained, can be used for comparison. The root loci are plotted on the s-plane because the variation of T causes the natural system roots to move on the z-plane. Crossing the unit circle into instability still corresponds to crossing the imaginary axis. Placing an arbitrary bound of 10% round the roots shows that we need about 160 samples/cycle when $M_q = .2$ and 21 samples/cycle when $M_q = 1.0$ to maintain the system simulation to this accuracy.

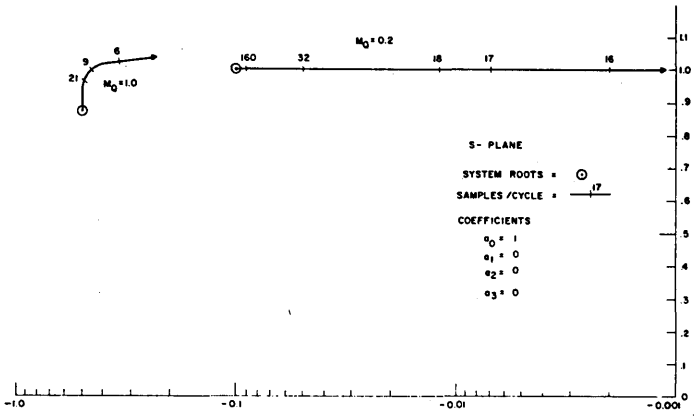


Figure 4—Root locus—no computation delay

Case II: No prediction but digital computation time corresponds to sample time, T, i.e.:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 1 \\ a_2 = a_3 &= 0 \end{aligned}$$

Figure 5 shows the root loci for the same two damping values. An extra root appears, but remains well up to the left on the real axis while the system roots move toward the imaginary axis and instability. Now 520 and 64 samples/cycle are needed to keep the roots within 10% of the correct position for $M_q = .2$ and 1.0 respectively.

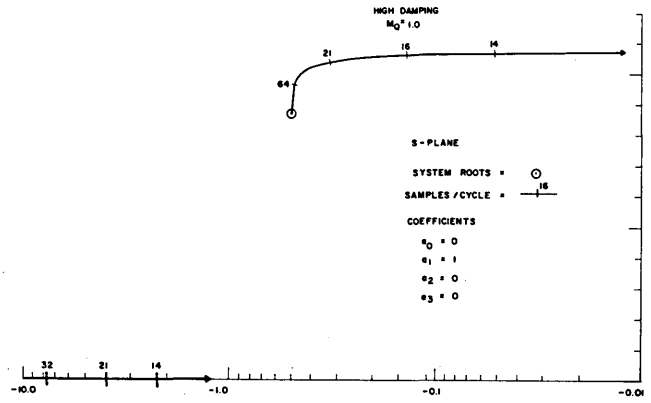


Figure 5(a)—Root locus—1 cycle computation delay

Case III: Applying the simple linear predictor to overcome the delay:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 2.5 \\ a_2 &= -1.5 \\ a_3 &= 0 \end{aligned}$$

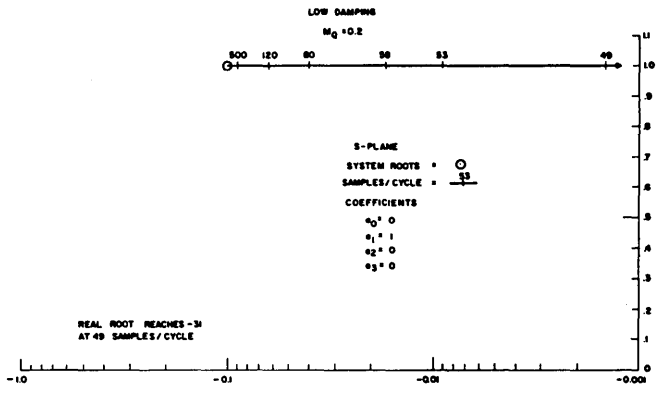


Figure 5(b)—Root locus—1 cycle computation delay

Figure 6 shows the corresponding root-loci. Of note now is the appearance of a second pair of roots since the system polynomial is now fourth order. For the low damping case, $M_Q = .2$, these extra roots stay over on the lefthand side, while the system roots move to the right, toward the stability limit. However, when we make $M_Q = 1.0$, so the basic system is heavily damped, the system roots move to the left; and the extra roots move to the right crossing the stability limit at 13 samples/cycle and producing an oscillation at 2.7 rads/sec.

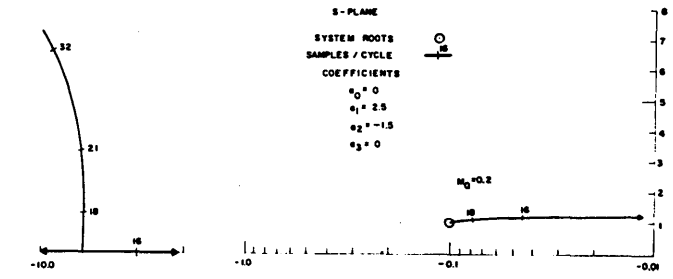


Figure 6(a)—Root locus—first order predictive compensation

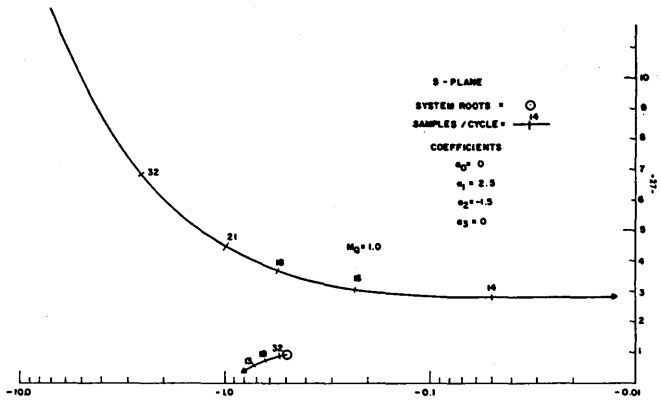


Figure 6(b)—Root locus—first order predictive compensation

Case IV: Using the quadratic filter to predict ahead for 1.5 sample times, the coefficients become:

$$a_0 = 0$$
$$a_1 = 4.375$$
$$a_2 = -5.25$$
$$a_3 = 1.875$$

Figure 7 shows the corresponding root loci. The five roots provide three extra, two imaginary and one real. In both cases the stability limit is exceeded by the extra root pair crossing the imaginary axis—at 10 samples/cycle when $M_Q = .2$ or at 28 samples/cycle when $M_Q = 1.0$.

Note that prediction controls the principal roots in the high damping case in such a way that the movement is negligible.

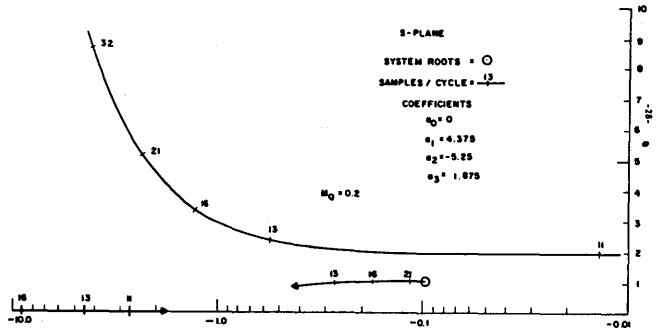


Figure 7(a)—Root locus—second order compensation

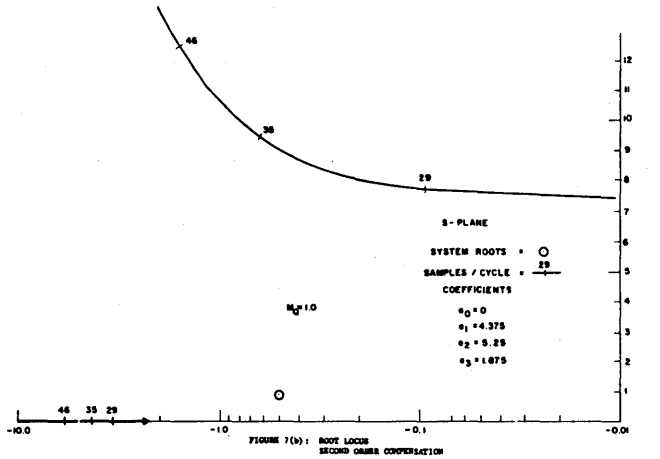


Figure 7(b)—Root locus—second order compensation

CONCLUSIONS

The application of predictive compensation to lightly damped systems, as suggested in the references, can result in accurate simulations for as low sampling rates as 10 per cycle. However, it must be pointed out that the important criteria for validity is whether we can apply this compensation to highly *stable* systems. In

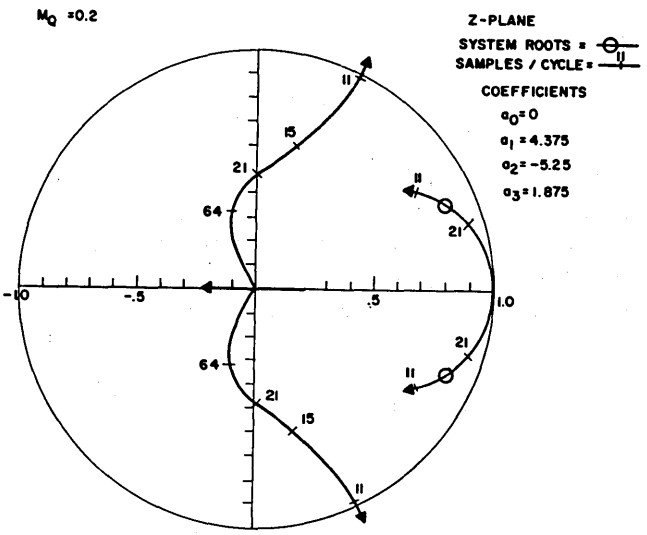


Figure 8—Root locus in Z-plane of second order compensation

fact, the majority of physical systems that come before the simulation engineer will be highly stable, since a normal design objective is to ensure suitable frequency response and adequate damping.

If digital compensation is to be applied to several systems—for instance as part of a software library—then no higher order than the first need be used and an adequate sampling rate must be maintained to keep the roots due to the compensating polynomial away from the region of interest. With unity damping (M_Q= —2.0) even first order prediction requires 25 samples/cycle for stability, and at least 30/cycle would be necessary for an adequate simulation. Basically, this is a restatement of the old requirement—that hybrid computation requires the fastest digital computation to maintain an adequate simulation performance.

A characteristic of these extraneous roots introduced by the prediction technique is the speed with which they move into the region of interest. That is, changing the sampling rate by 1 or 2 samples/cycle can make all the difference. Conversely, with a fixed sampling interval, no warning is given that stability limits are being approached, since a small change in natural frequency can cause the oscillation to appear.

Appendix: Filter Coefficients

Munroe[8] has shown that the coefficients $a_0 - a_n$ to pass a polynomial of degree q without distortion can be obtained from solution of the constraint equations:

$$\sum_{k=0}^{N} a_k = 1$$

$$\sum_{k=0}^{N} k\, a_k = -\alpha$$

$$\vdots$$

$$\sum_{k=0}^{N} k^q a_k = (-1)^q \alpha^q$$

where α is the number of time steps extrapolated—fraction or integer, i.e.,

$$M_{j+\alpha} = \sum_{k=0}^{N} a_k M_{j-k}$$

If α is negative, then an interpolation is obtained which is useful for data smoothing or reconstruction. If $q = N$, then the a's are completely defined, but if $q<N$, then the a_k's are chosen to minimize $\Sigma\, a_k^2$ which is the noise amplification by the filter.

REFERENCES

1 T MIURA J IWATA
Effects of digital execution time in a hybrid computer
Proc FJCC 24 251-66 1963

2 W J KARPLUS
Error analysis of hybrid computer systems
Simulation 6 121 1966

3 E G GILBERT
Dynamic error analysis of digital and combined analog-digital computer systems
Simulation 6 241 1966

4 D L MATLOCK
Pulse prediction filters applied to digital and hybrid simulation
Simulation 6 163 1966

5 R M DEITERS T NOMURA
Circle test evaluation of a method of compensating hybrid computing error by predicted integral
Simulation 8 33 1967

6 R GELMAN
Corrected inputs—a method for improved hybrid simulation
Proc FJCC 24 267-276 1963

7 M CONNELLY O FEDOROFF
A demonstration hybrid computer for real time flight simulation
AMRL-TR-65-97

8 A J MUNROE
Digital processes for sampled data systems
Wiley 1962

Hybrid Apollo docking simulation

by BRUCE JOHNSON
NASA Manned Spacecraft Center
Houston, Texas
and
SAMUEL S. WEINER
Lockheed Electronics Company
Houston, Texas

INTRODUCTION

The Apollo manned lunar landing program of the National Aeronautics and Space Administration has as requirements the execution of various rendezvous and docking maneuvers in space by the command and service module (CSM), the lunar module (LM), and the S-IVB space vehicles. To insure the success of the mission, the following four docking maneuvers are required.

1. Earth orbital
2. Translunar
3. Lunar orbital
4. Lunar separation

These maneuvers involve several different vehicle configurations exhibiting very different dynamic characteristics. The docking hardware assemblies (Figure 1), designed and fabricated by North American Aviation, Inc., consist of an active probe mounted on the CSM and a cone-shaped drogue mounted on the LM. The docking system hardware was designed to perform two essential functions:

1. During docking in space, the docking system attenuates the energy between the two vehicles and, after achieving a latched configuration, draws the vehicles together to form a tunnel for the transfer of personnel and equipment.
2. The docking system allows safe separation of the two vehicles during lunar orbit.

The purpose of the docking simulation was to flight-qualify the probe and drogue hardware prior to their use during an actual mission. These subsystems had previously been evaluated by all-analytical simulations and had been subjected to three-degree-of-freedom, all-physical simulations using air-bearing vehicles. However, because of the critical importance of the four docking maneuvers to a successful lunar landing mission, NASA decided to conduct full six-degree-of-freedom tests on actual flight hardware under a simulated space environment. The following are the objectives of this test program:

1. To demonstrate the structural capability of the docking system and its ability to attenuate the energy of the closing vehicles.
2. To demonstrate the latching capability of the system under a variety of conditions.

A detailed evaluation of the control aspects of the docking maneuvers was not included in this study.

Problem characteristics

The Apollo docking maneuvers resemble very-lightly-damped oscillators. The rigid-body natural frequencies are 2 to 3 cps with frequencies of interest ranging up to 10 cps. Reaction-control-system (RCS) frequencies range up to 80 cps. The assumption was made that the target vehicle RCS had no initial error, which then allowed the docking maneuvers to be simulated with six relative degrees of freedom. Only five axes were dynamically significant but, the sixth axis (roll) was included primarily for the purpose of simulating initial roll angles.

For this test program, simulation tolerances of ± 0.040 inches radially were specified at the latching rings. For one docking configuration (transposition) the distances from the vehicles center of gravity to the docking mechanism are 375 and 165 inches respectively. In that case, a radial tolerance of ± 0.040 inches is 0.007% of full scale. Figure 2 is a graphic representation of the vehicles showing this relative accuracy requirement.

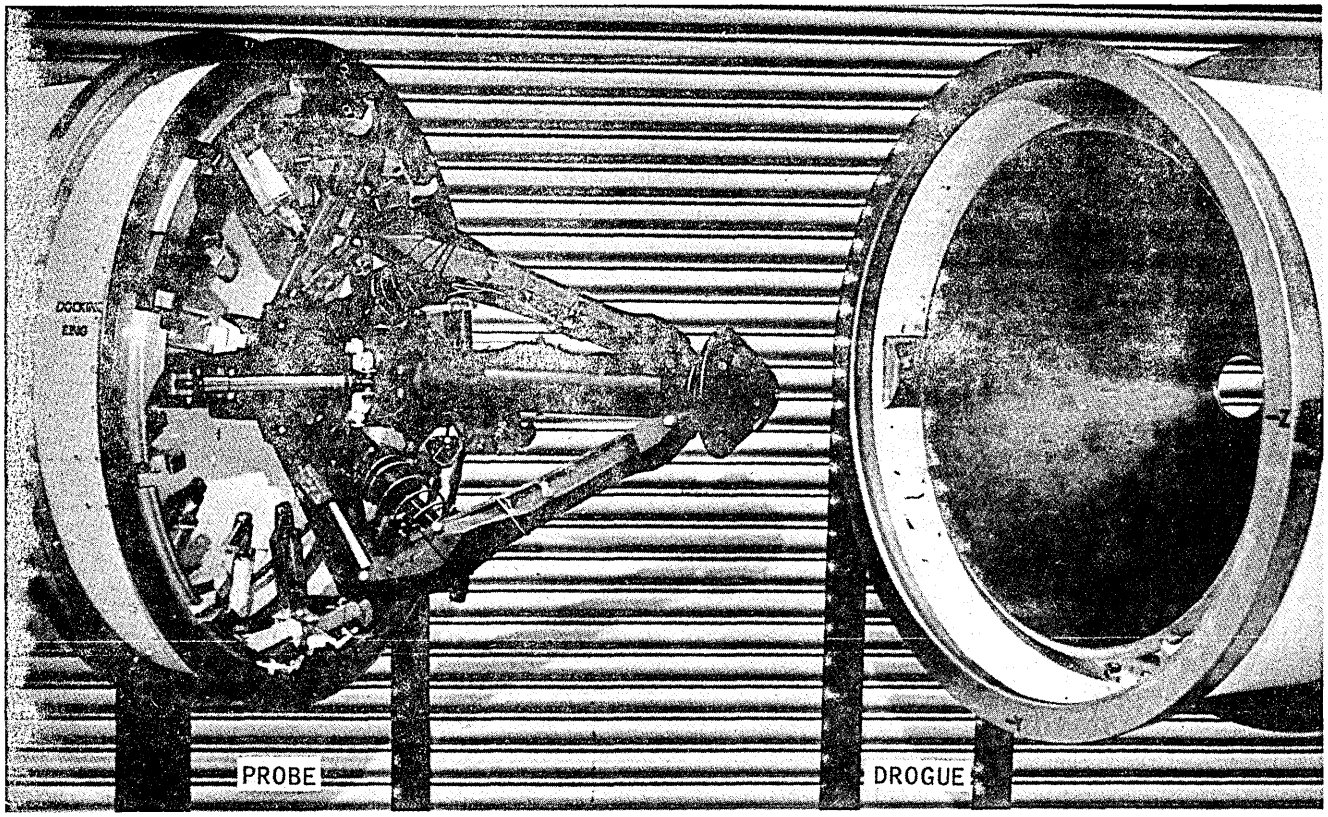
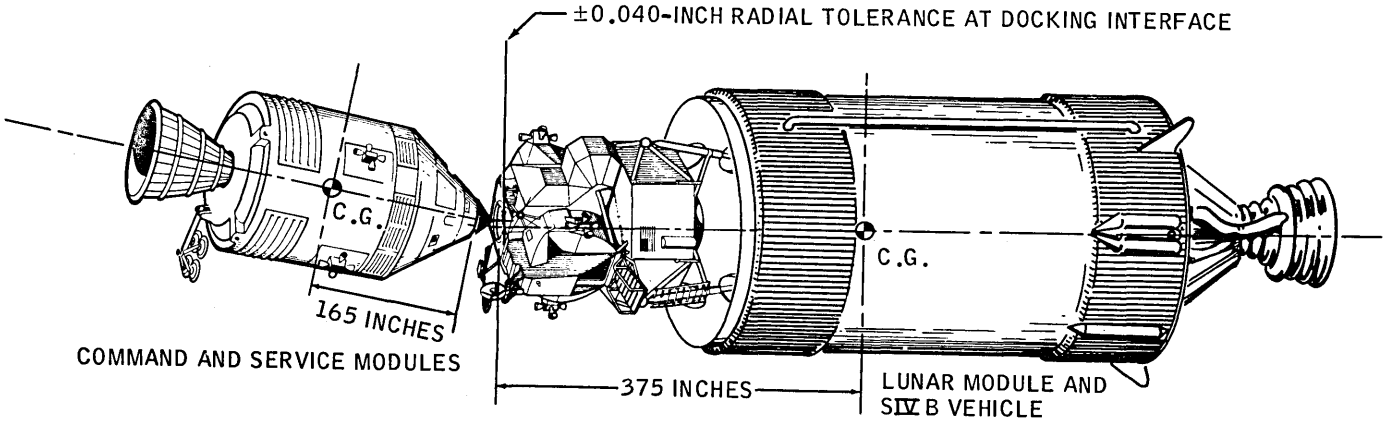


Figure 1—Apollo docking hardware



$$\text{INTERNAL ACCURACY} = \frac{0.040 \text{ INCH}}{165 \text{ INCHES} + 375 \text{ INCHES}} = 0.007\%$$

$$\text{COMMAND SIGNAL ACCURACY} = \frac{0.040 \text{ INCH}}{20 \text{ INCHES (FULL SCALE)}} = 0.2\%$$

Figure 2—Relative accuracy requirement

Simulator concept

Since the masses of the vehicles are large compared to those of the docking system components, it was necessary to simulate the rigid-body dynamics. To achieve realistic motion it was necessary to include a simplified functional simulation of the control systems. Furthermore, elastic-body bending and fuel slosh have frequencies within the range of interest. To test the docking system, NASA devised the Apollo Docking Test Device (ADTD) concept (Figure 3). The probe and drogue hardware are mounted on the docking rings, and each ring is supported by six load cells. The load cells are mounted on two structures which can be moved hydraulically. The probe side of the ADTD has two degrees of freedom (X-axis and roll axis). The remaining four degrees of freedom are on the drogue side of the ADTD. Using loads measured by the ADTD, a computer simulated the vehicle dynamics and fed position commands back to the hydraulic actuators.

The ADTD was built by American Machine and Foundry Company. It weighs 50 tons, largely because of the seismic-mass base plate. The six main hydraulic actuators of the ADTD are controlled by position servo-mechanisms and have a frequency response of 12 cps. Riding "piggy back" inside the main translational actuators are high-frequency, low-amplitude actuators for the simulation of elastic effects. The ADTD is capable of operation in a 10^{5}-ton environment, in temperatures of —80° to 250° F.

In addition to the hydraulic and load-cell systems, the ADTD incorporates quick shutdown or abort logic for the protection of the personnel and of the docking system. The abort system has inputs from the computer; from the operator, and from circuits monitoring the motion envelope, the forces, the velocities, and the strains and impulses.

The computer

The computer used for the docking system simulation was an EAI 8900 hybrid system (Figure 4). In explaining why a hybrid computer is used, the engineer is normally placed on the defensive. In this case, the simulation engineers felt that the hybrid computer was the best tool for the job. To justify this position, it is necessary to explain why an all-analog or all-digital computer system could not have been used.

In 1964, when this work was begun, there was no

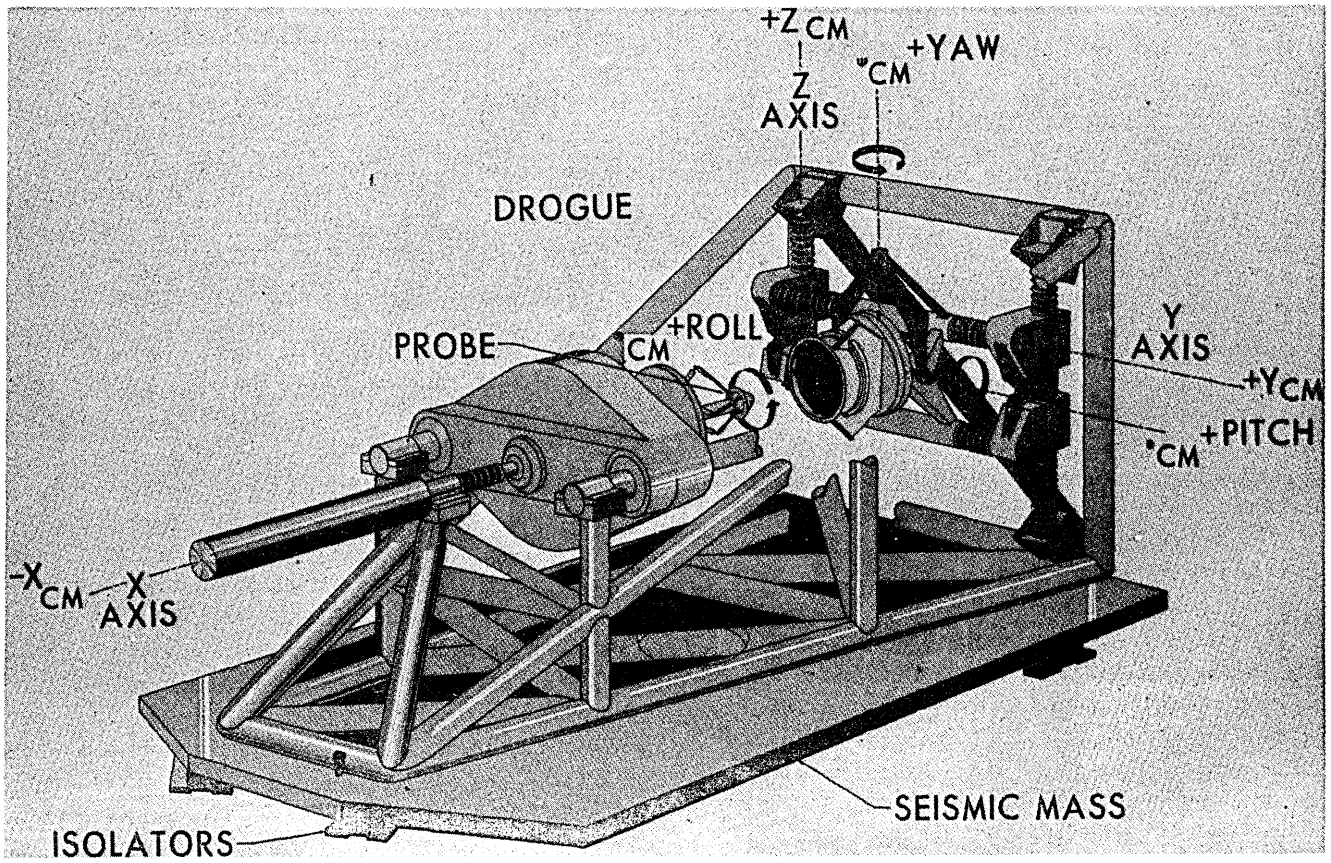


Figure 3—Apollo docking test device, perspective

Figure 4—The EAI 8900 hybrid computer system

digital computer available which could do the work in real time. Furthermore, since the docking simulation was to operate on two shifts per day, a large digital computer would have been prohibitively expensive.

An all-analog simulation was given serious consideration. In fact, the docking simulation employed an interim analog computer system, which consisted of three EAI 231 R computers, a DOS 350, and an ADIOS. While the use of the interim analog computer was very beneficial in debugging the mathematical model, in solving computer-hardware interface problems, and in familiarizing personnel with the problem, it nevertheless demonstrated conclusively that an analog computer could not solve the problem satisfactorily.

The most pertinent reason for abandoning the analog simulation was accuracy. As explained earlier, this simulation required small differences of large numbers. The required resolution was outside the range of a single amplifier, not to mention the overall resolution of a 400-amplifier problem.

Of equal importance was the need for (1) fast turnaround, (2) convenient setup, (3) virtually foolproof operation, and (4) extensive documentation. Figure 5 is a summary of the analog procedures necessary for operations and indicates approximate times for each step. Because of the flight-qualification nature of the project, all operator functions were contained in a written procedure or countdown. For the analog simulation, 5 days of leadtime was needed to allow for certain changes which required off-line support of two digital computers. A setup and static check of the analog simulation required a minimum of 3 hours. Changing docking configurations also required 3 hours. Since these operations were extremely time consuming, they could not be performed very often. There was also considerable uncertainty as to what problem the computer was actually solving, since production-run documentation was virtually nonexistent.

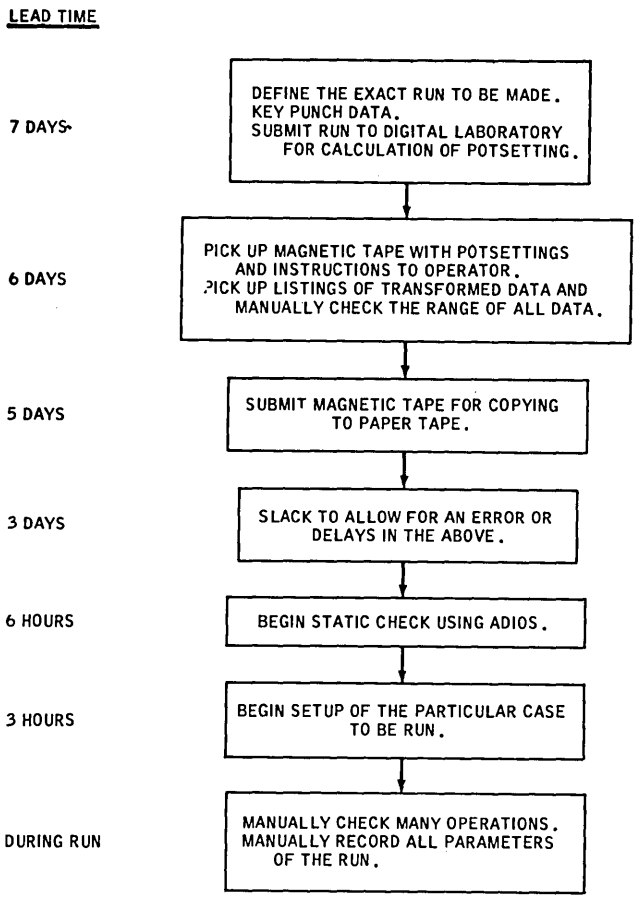


Figure 5—Summary of analog procedures

Essentially, use of the hybrid computer system removed the operator from the loop. The operator ran the simulation from an interpretive control program. A complete setup and static check was performed in 10 minutes, exclusive of any troubleshooting required but including all operations formerly done off-line by another computer. The results of all operations were documented in a very readable form which reflected the actual state of all problem parameters. Having this capability, static checks were performed with comparative frequency, thus increasing overall confidence in the simulation. Figure 6 is a summary of comparable information and times for the hybrid system. The hybrid system also permitted the incorporation of real-time, on-line diagnostics and debugging aids. For example, this program could supply simulated forces to permit dynamic checking of the problem, without the ADTD, on an open-loop basis. During a test, the

computer continually monitored error conditions such as analog computer overloads, A/D and D/A converter over-ranges, timing failures, and erroneous operator commands. If an abort occurred in the ADTD or in the computer, an analysis of the abort cause was performed, and the offending condition was documented automatically. Figures 7 and 8 are examples of the documentation which was extremely necessary for this simulation.

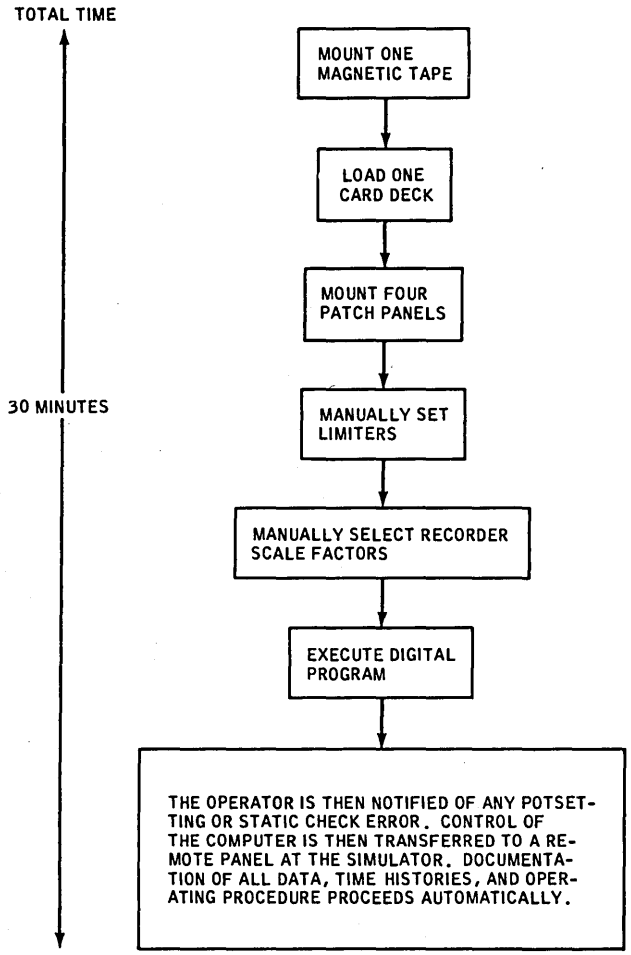


Figure 6—Summary of hybrid procedures

Availability was also a major consideration in using the hybrid computer. The 8900 system was procured primarily for this simulation, but it was also necessary to use a general-purpose computer system which could also solve other types of problems encountered in the Apollo Spacecraft Program.

The simulation

Two mathematical models were used in the simulation. The first was an open-loop algebraic model used only for initialization. It will be discussed later. The second model was a six-degree-of-freedom, rigid-body representation of two vehicles. It received inputs from the load cells and developed forces, moments, and body rates for each vehicle. Relative velocities were then computed and integrated to achieve relative displacements. A final tranformation converted the relative displacements to the ADTD coordinate system. The model also included a two-stage reaction control system for each vehicle. Provisions were made for the addition of elastic-body-bending and fuel-slosh models. Figure 9 represents the major blocks of the total mathematical model (Reference 2).

The model was formulated by expanding the equations of motion for each body in its axis system and then describing the relative motion in the axis system of one of the bodies. To eliminate unnecessary calculations, axis systems and rotational sequences were chosen which would exploit the ADTD capabilities. Since the equations were formulated in one body axis system, it was unnecessary to convert the motion of each vehicle to an inertial system, to calculate the relative motion, and then to transform the relative motion into the reference body system. This reduced the number of transformations from 12 to 7 and improved the solution accuracy.

After the decision was made to use a hybrid computer system for the simulation, it was necessary to analyze the problem in detail to determine the best distribution of computation to be programmed for each computer. The following factors were considered:

1. Real-time solution constraints
2. Computation accuracy requirements
3. Economy of equipment
4. Ease of operation
5. Computer-ADTD interfacing
6. Computation frequencies

After imposing the aforementioned considerations, the model was programmed with the force and moment resolutions, the body rate computations, and the RCS implemented on one analog computer and the force abort system implemented on the other analog computer. The digital computer calculated the relative velocities, relative positions, and the actuator command signals.

This particular distribution very naturally satisfied the previously mentioned considerations. The digital computer was capable of handling its function in real time. It updated at 100 frames per second, giving 40 samples per cycle at the natural frequency of the system and 10 samples per cycle at the highest frequency. The relatively higher frequencies of the forces, the moments, and the control system were programmed

```
MORRISON        3/05/67  11:13:46
    TIME=   0.000000E 00
    LX  =   -0.152250E 03      LY  =  -0.789999E 01      LZ  =   0.600000E 00
    LX  =    0.538700E 02      LY  =  -0.300020E 00      LZ  =   0.200000E 00
    MASS1 =  0.100500E 04      MASS2 =  0.178000E 03
    IXX =  0.177598E 05      IYY =  0.549680E 05      IZZ =  0.560200E 05
    IXY = -0.456500E 04      IXZ =  0.480100E 04      IYZ = -0.146100E 04
    IXX =  0.283900E 04      IYY =  0.261000E 04      IZZ =  0.159800E 04
    IXY = -0.759999E 01      IXZ =  0.550000E 01      IYZ =  0.240000E 01
    V=   -2.21      V=    0.40      V=   -1.40
    V=    0.05      V=    0.04      V=    0.04

RUN TERMINATED BY FORCE ABORT FX2  AT T =  10.60      3/05/67  11:17:53
    IMPACT AT TIME = 10.00
    DRIVE -0.0749 -0.0452 -0.0449  0.0894 -0.0857 -0.6921  0.3002  0.3058  0.0000  0.0000
           0.5995  0.2656 -0.2660  0.5758 -0.6106  0.6252 -1.0578  0.0500  0.0510  0.0000
    FREQUENCIES     0.1000E 00     0.1000E 00     0.1000E 00     0.1000E 00     0.1000E 00     0.1000E 00
    THE FOLLOWING DATA AT TIME = 10.00
    V=   -2.21      V=    0.41      V=   -1.40
    V=    0.05      V=    0.06      V=    0.06

    W=   0.0000      W=    0.005      W=    0.017
    W=  0.00010      W=  0.00002      W=  0.00003
    EJLDOT= -0.153299E-02      EULDOT=  0.121520E-01      EJLDOT= -0.122463E-01
```

Figure 7—Line printer output

```
   $JOB   V14       PRODUCTION    CONTROL DECK
ENTER NAME.
MORRISON
PUT 8800 UNDER I/O COMPUTER CONTROL FOR MODES AND ADDRESSES.
PRESS FLAG.8 TO CONTINUE.
RCS = F
CKOUT = T
ENTER OPTION NUMBER AS FOLLOWS, THEN CARRIAGE RETURN
      1 = STATIC TEST
      2 = CONFIGURATION SETUP
      3 = RUN SETUP
      4 = INITIALIZATION
      5 = IDLE WITH INTERRUPTS ENABLED
      6 = TOGGLE RCS
      7 = TOGGLE CKOUT
2
SEQUENCE ERROR
LOAD CONFIGURATION DATA
NEXT OPTION PLEASE
3
ARE COEFFICIENTS FOR THIS RUN TO BE COMPUTED OR READ FROM CARDS?
      1 = COMPUTE
      2 = READ CARDS
1
RUN SET UP COMPLETE.
NEXT OPTION PLEASE
4
INITIALIZED.
RUN TERMINATED BY LIMIT IN SPIT    AT T =  14.21      3/05/67  11:13:26
INITIALIZED.
RUN TERMINATED BY LIMIT IN SPIT    AT T =  14.26      3/05/67  11:14:13
INITIALIZED.
RUN TERMINATED BY MANUAL OPR ABORT AT T =   1.79      3/05/67  11:15:31
INITIALIZED.
RUN TERMINATED BY FORCE ABORT FX2  AT T =  10.60      3/05/67  11:17:53
INITIALIZED.
```

Figure 8—Typewriter output

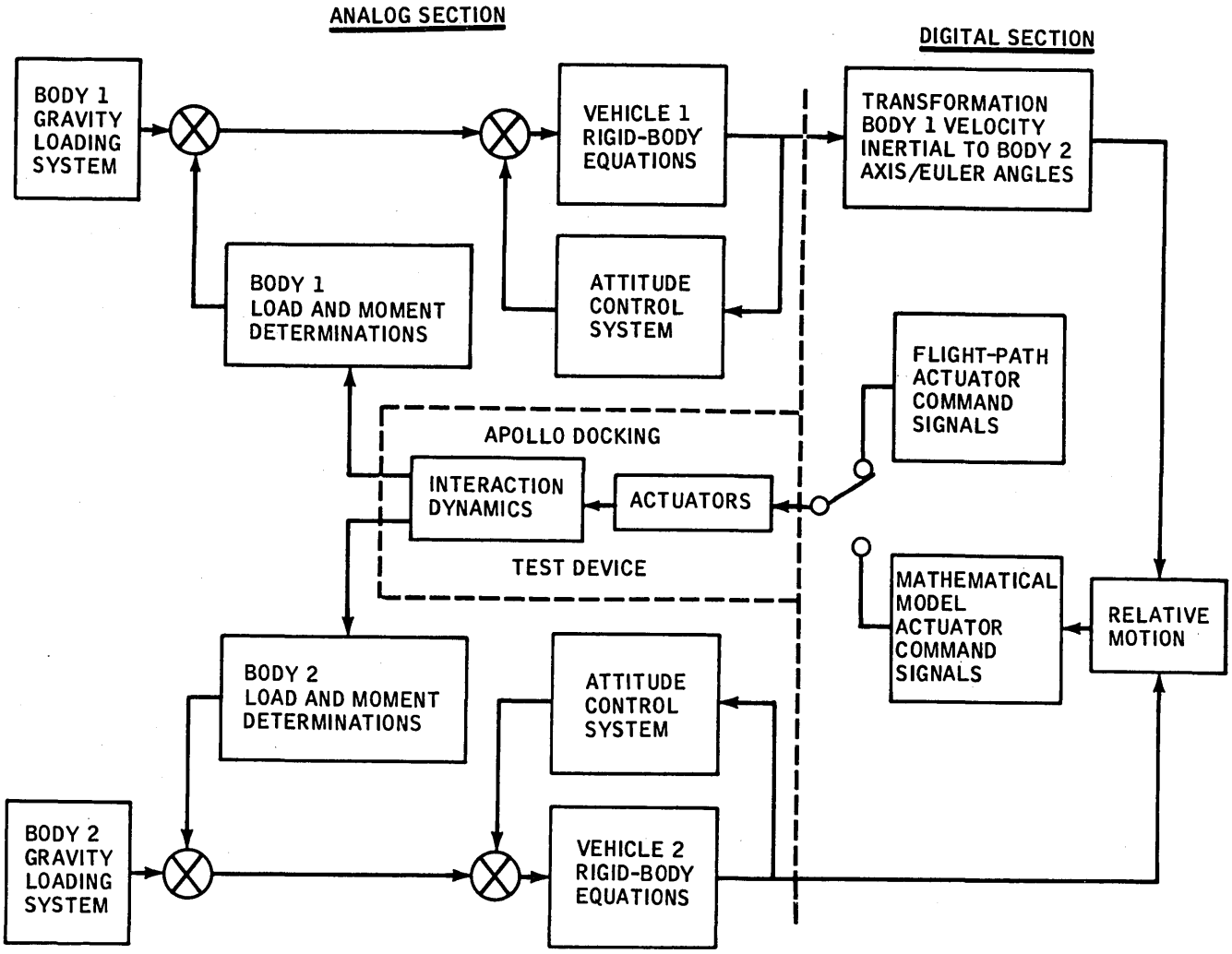


Figure 9—Simulation block diagram

on the analog computer. Figure 10 is a time history of the forces and moments of a representative docking run.

The accuracy problem arose when the relative body motions were transformed to the reference points in the probe and in the drogue. To overcome the problem this computation was programmed on the digital computer.

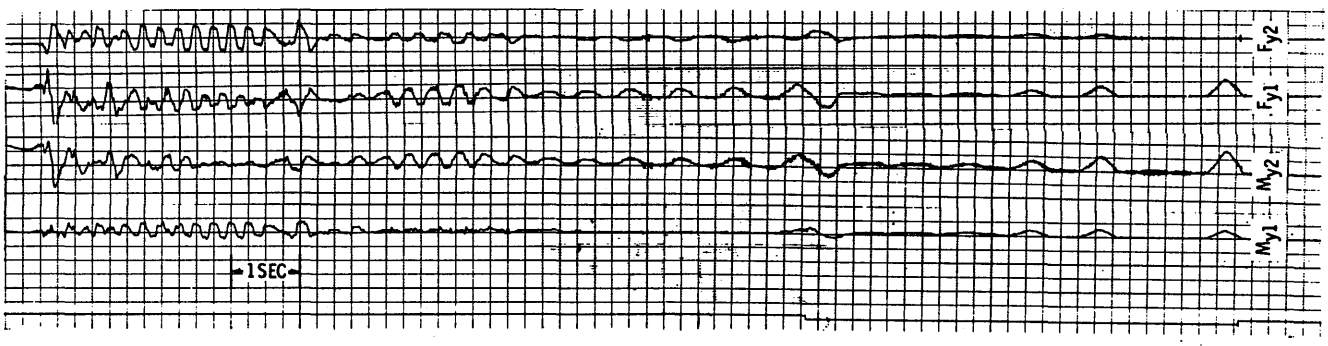


Figure 10—Time history of forces and moments

The available equipment was employed very economically since a minimum of interfacing equipment was required. Only 12 A/D channels were required for the velocities, and six D/A channels were used for the hardware command signals. Fourteen other D/A converters were used for data acquisition. Furthermore, since all transformations were performed digitally, no resolvers were required, and the number of multipliers was minimized.

Implementation of the dynamic model was 75% of the analog task but only 25% of the digital program.

The following tasks were also integral to the simulation:

1. Setup, checkout, and limited automatic scaling capability—A full static check required about 3 minutes; a complete initial setup or change in docking configuration required 5 minutes. Because of computer requirements on other projects, a goal of 30 minutes notice from cold start to operational readiness was established. This goal was achieved by automating in every possible area.

2. Real-time control logic and dynamic error checking—To prevent damage to the hardware and to prevent unintentionally aborted runs caused by operator errors, sequencing checks were made on all control functions. Control logic permitted local operation at the computer or fully remote operation from the simulator console. Dynamic checks were provided for computer problems such as overrange, overloads, excessive A/D offsets, and interrupt timing errors. Fully automated control of high- and low- speed data-acquisition systems was also included.

3. Development and calculation of flight trajectories. —The open-loop algebraic mathematical model (referred to earlier) was developed only to achieve the specified initial conditions for a particular docking run. The customer for this project (another NASA division) specified the different test cases by assigning miss distances, misalinements, and relative rates at the time of contact of the probe and drogue. Since the customer was only interested in contact and postcontact data, the method used to reach these "initial conditions" (IC) was arbitrary within the acceleration capabilities of the ADTD. The method chosen to achieve the specified initial-contact conditions was (1) to always start from the same reference position with all axes at zero (with the exception of the X-axis) and (2) to "fly" the hardware to contact while, at the same time, achieving all the specified contact conditions. The development of a set of equations to fly the ADTD hardware and, simultaneously, to establish the proper contact conditions was based upon complete knowledge of the ADTD performance capability.

To fly the simplest path by analytical means and to satisfy the boundary conditions at time = 0 and time = T, third-order equations in time were necessary. The cubic equations were used to drive the hydraulic actuator until contact. The derivation of the general equation follows. Let y(t) represent any of the desired actuator positions such that y(T) and y(T) are obtainable and are related to the specified terminal conditions given at time T (where T = time of contact of probe and drogue when all preassigned conditions are met). A cubic equation in t can now be represented as

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \tag{1}$$

but $y(0) = 0$ and $y(0) = 0$; therefore,

$$\left.\begin{aligned} y(t) &= a_y t^3 + b_y t^2 \\ y(t) &= 3a_y t^2 + 2b_y t \end{aligned}\right\} \tag{2}$$

where $0 \leq t \leq T$. At the terminal time T

$$y(T) = a_y T^3 + b_y T^2 \tag{3}$$

and

$$y(T) = 3a_y T^2 + 2b_y T \tag{4}$$

Simultaneous solution of equations (3) and (4) for a_y and b_y yields

$$a_y = \frac{Ty(T) - 2y(T)}{T^3} \tag{5}$$

and

$$b_y = \frac{3y(T) - Ty(T)}{T^2} \tag{6}$$

Equation (2) was represented n times for n degrees of freedom.

Because of the initial bias on one axis (X = 25 inches) at t = 0, calculation of the coefficients was slightly different for the X translational terms and must be calculated as follows:

$$a_x = \frac{Tx(T) - 2[X(T) - 25]}{T^3} \tag{7}$$

$$b_x = \frac{3[X(T) - 25] - Tx(T)}{T^2} \tag{8}$$

The digital setup program transformed the desired contact conditions into the body 1 reference frame and set the initial conditions on the body 1 integrators. The body 2 initial velocities were always zero. The setup program also computed the coefficients of the

six subic equations which describe the flight path. An arbitrary flight time of 10 seconds was normally used. For a test run, the digital computer evaluated the six flight equations in real time until hardware contact, at which time the rigid-body mathematical-model computation began.

Smooth switching between the algebraic flight model and the dynamic model was one of the more difficult tasks in the simulation. At the time of contact, several things happened simultaneously. The analog computer switched from "IC" to "Operate," and the control system was activated. The digital integrators were also initialized using past derivatives computed during free flight. If contact occurred at other than 10 seconds because of system tolerances, a first-order discontinuity occurred. This program was further compounded by leads built into the digital program. These problems will be discussed later in the paper.

The following are other incidental, but very important, features of the program.

1. Extensive logic was provided to assist in trouble-shooting. The entire program could be operated in a checkout mode.
2. A force abort system monitored the absolute value of four resolved forces and aborted the test when necessary to protect the hardware.
3. Every test run was automatically documented in considerable detail.
4. XZ and YZ displays were developed for the operator.
5. Upon occurrence of an abort, available data were analyzed to determine the probable cause.

Major problem areas

A project of this magnitude naturally generated numerous problems. Some of these problems and their solutions are of general interest. Visitors to the simulation laboratory first notice the unusual environment under which these tests are conducted. The computer and the simulator are housed in different buildings separated by about three city blocks. One-way video and two-way voice communications are used to establish contact betwen the two sites. Because of the flight-qualification nature of the tests, all procedures were written into a countdown which was controlled by a test director. The communications coming over the loudspeakers gave the tests a science-fiction aura; however, because of (1) the high energies stored in

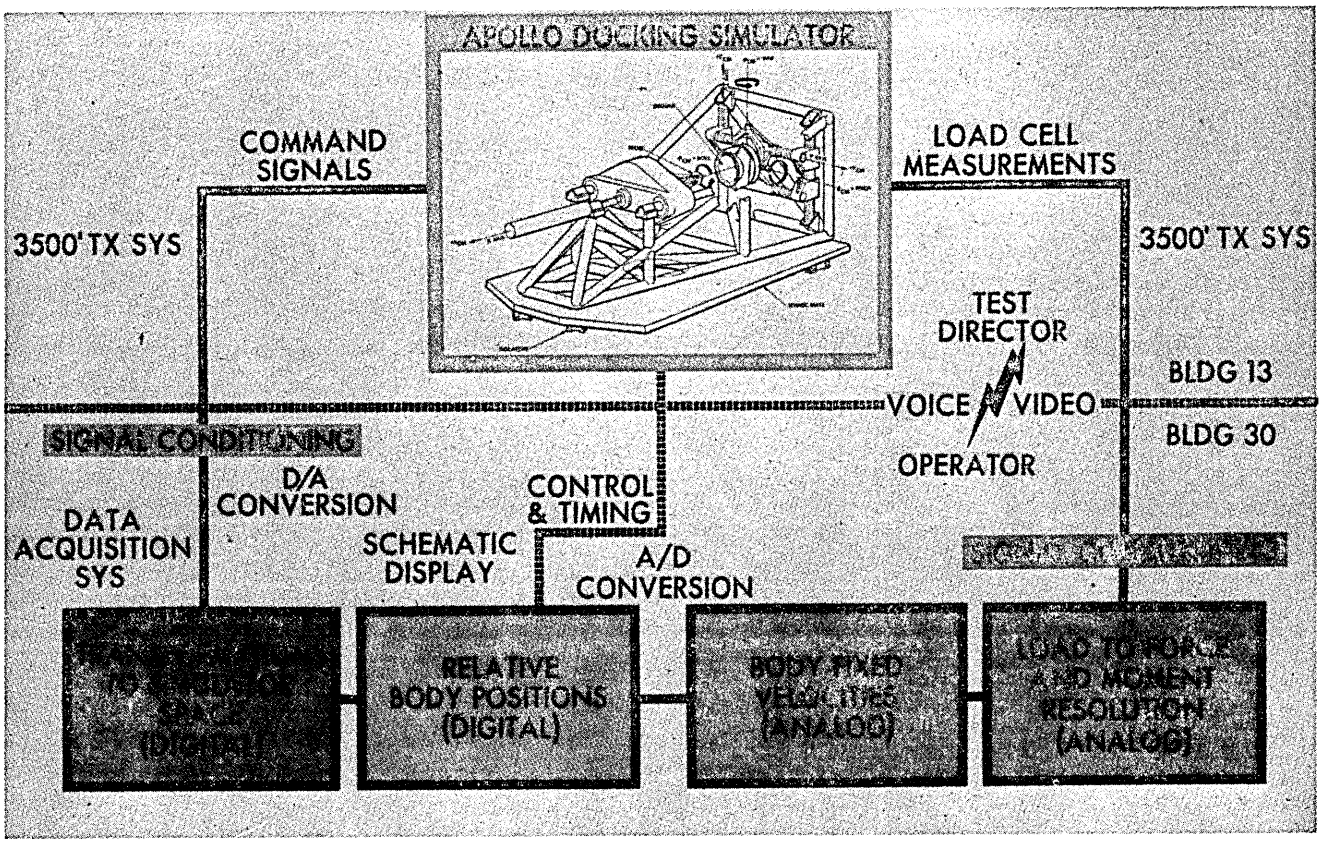


Figure 11—Real-time Apollo docking hardware tests system —overall system diagram

the system, (2) the cost of the facilities, and (3) the large number of personnel on the test operations team, this type of coordination was very necessary. Figure 11 is a graphic presentation of the overall system diagram.

Visitors also note the plexiglas covers and seals on the analog patch boards. The digital program decks, listings, and tapes are similarly sealed. This is a quality control requirement. This sort of control was very awkward at first and was a most unnatural environment for the computer programmers, who felt severely handicapped. However, after several months of operation under these conditions the advantages of reliable documentation became apparent. The files now contain a complete record of every change made since October 15, 1966. Confidence in the accuracy of the documentation and the overall simulation is very high.

The development schedule of this simulation was also well documented and is of interest. The actual programming and checkout was accomplished in 2½ months by three men, working 16 hours a day, 7 days a week. The job ended on October 15, 1966, with quality control acceptance of the work. The job might have been completed earlier, except for the multiple handicaps of a new and untried computer with new, untried software being used by personnel who learned to use the machine while programming this simulation.

Approximately 4 months of complete system checkout followed the October 15 acceptance date. The most significant system problem was stability. The system exhibited divergent oscillations for the light-vehicle tests during lunar-orbital docking as the two simulated vehicles neared a fully-docked position. Analysis of this problem revealed that as the probe retracts, its stiffness is greatly increased, which causes the natural frequency to be increased and causes the system lags to approach 90°.

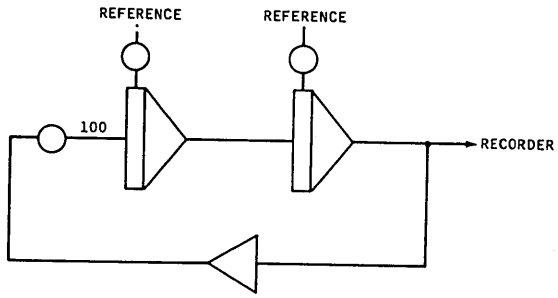
An analysis of the system componets revealed 12° of lag in the ADTA servomechanism and 15° of lag caused by the digital computer at the frequency of interest (2.5 cps). Several things were done to eliminate these lags.

1. The ADTD servosystems were tuned to improve performance in the range of interest, at the expense of higher frequencies.

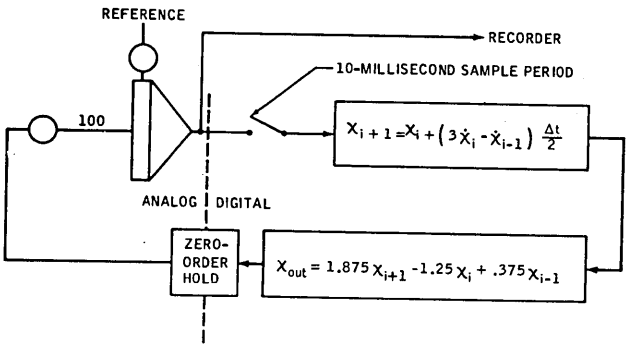
2. The digital integrator was investigated. The original integrator was a simple Euler algorithm. For the investigation it was packaged in a separate program which duplicated the timing of the full simulation. The package operated as a simple integrator with a single gain-one input and one noninverted output. With this circuit, integrators were studied for the simulation which, as previously mentioned, resembled very-light-damped oscillators. The stability of the all-analog loop was first verified to qualify it as a standard. The EAI 8800 oscillator was extremely stable in the range of frequencies studied (0.1 to 20 cps). Next, one analog integrator was replaced with the Eulerian integrator. As expected, the loop was unstable at 1 cps with a step size of 0.01 second. Then a polynomial lead of $1.5\Delta t$ was added to the digital routine (References 1, 3, 6, 7, 8, and 9). This is the theoretical, demonstrated value used to correct for a pure digital lag with a zero-order hold on the output. The oscillator stability was improved but was still not acceptable.

The final step was to use a first-order-predictor integrator and a pure time lead of $0.5\Delta t$ on the output. With this configuration the loop was very stable up to approximately 9 to 9.5 samples per cycle (11 cps). At the time the integrator was changed, the ADTD was exhibiting a pure timelag of 6°/cycle. To compensate for this an additional 16.66-millisecond lead was added to the equations, which still had undesirable but acceptable oscillations. Figure 12 represents the integrator evaluation circuits used for the study. Figure 13 is a comparison between the two oscillators.

3. The transmission system, as mentioned previously, was operated remotely. The cable run was 3500 feet through utility tunnels carrying major power trunks. The analog information was carried as ±10 V dc, and control information was +28 V dc and 0 V dv.



(a) Experimental control circuit for comparison – all analog.



(b) Hybrid oscillator being evaluated.

Figure 12—Integrator evaluation circuits

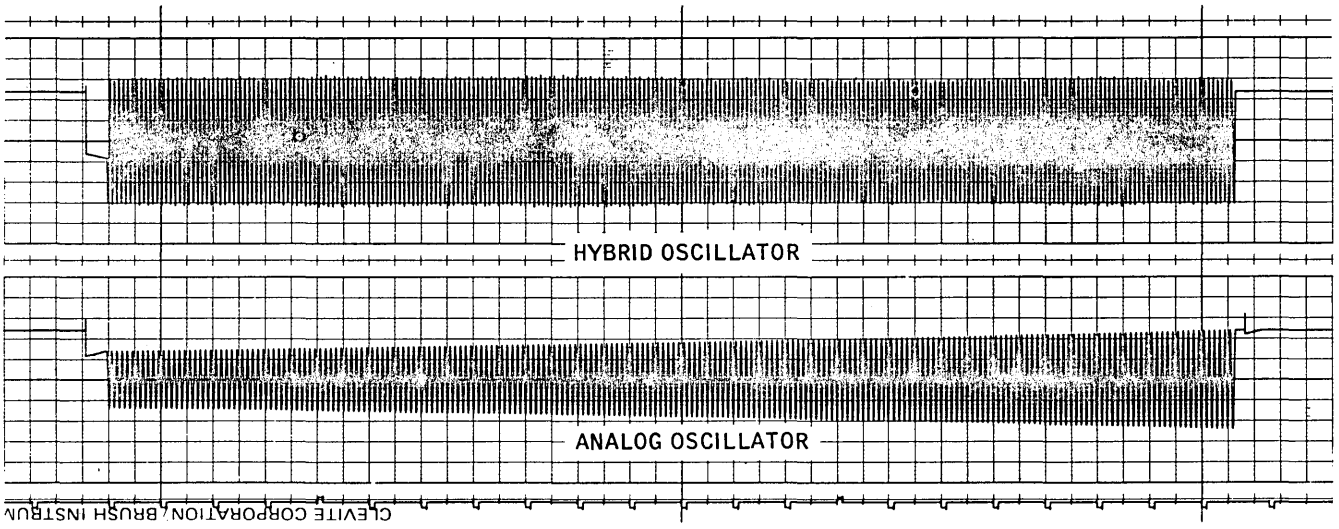


Figure 13—Analog and hybrid oscillators, 10 cps

Grounding and noise rejection were early problems of concern, but the first system tested for control signals worked very well. Relays were used for isolation, and the commons of the relay power supplies were tied together at each end. Results were not as good for the analog signals. Three systems were tried before a final configuration was reached.

The first was a chopper-driven, transformer-coupled system built by AMF. The phase lags and noise produced by the choppers were excessive. The second method used differential amplifiers. Noise levels were still intolerable. Finally, the transmission terminal equipment was completely eliminated. The lines were then driven, with no problems, directly from operational computing amplifiers. Considerable 60-cycle noise was picked up because of ground potential differences. On input, this was removed by subtracting the signal on the line (which was grounded at the simulator end) from all of the other signals. On output, noise is added to the command information before transmission. Figure 14 is a schematic diagram of the common mode rejection circuits (References 10 and 11).

The evolution of setup and checkout philosophy on this project was very enlightening. The first approach used hand-calculated potsettings and static check values stored on punched cards. It was soon evident that the frequency of changes to these decks would create an excessive workload.

A program to calculate potsettings was written, and the flexibility of changing problem parameters was greatly improved. However, the program was so large that changes to it were difficult. A single potsetting change required about 2 hours for recompilation and loading, but this method was used nevertheless because of reliability and speed of calculation. The reliability stemmed from the fact that data could be loaded into the computer once and used for potsettings, digital parameters, and documentation. This eliminated the inherent unreliability of having a person input the same data twice.

The static check philospophy was generalized somewhat by making it interpretive and by causing it to recognize all forms of A/D communication. However, it was still manually prepared, difficult to maintain, and hardware oriented. Memory space was at a premium, largely because of the size and complexity of the setup programs. As the system monitor grew in size, it became necessary to overlay programs. The simulation had two overlays which were automatically swapped when required. Setup and static checkout was in one overlay while initialization and all real-time programs

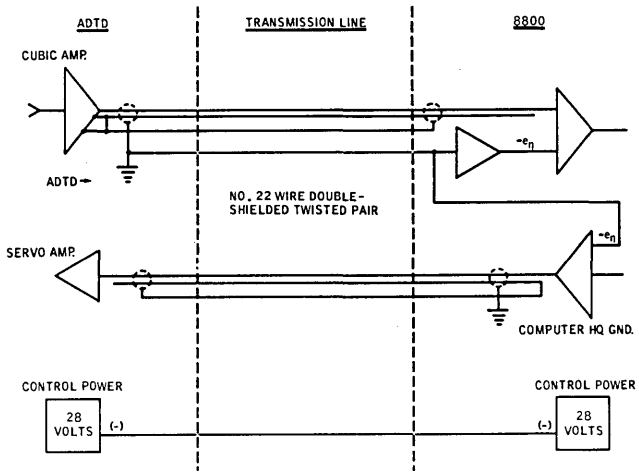


Figure 14—Common mode rejection circuits

were in the other. The data area was common to both overlays.

CONCLUDING REMARKS

If this simulation were to be restarted with present experience, several significant changes would be made. The functions of potsetting calculations and static checkouts would be performed by a problem-oriented interpreter. This would permit rapid modification to the analog program, with corresponding setup program changes. It would also permit a calculated static check for every configuration, rather than the single hypothetical case which was used. Such an interpreter would look very much like the EAI Hytran Operations Interpreter, but three very significant changes would be made. The interpreter would be subordinate to the main control program of the simulation, and it would communicate with the rest of the simulation by means of a symbol table. Without these features, the operator must input the same data twice in different formats. The interpreter would also have the ability to handle multiple analog computers. In addition, a fully automatic scaling program would be written for the analog and conversion system programs.

The overall development schedule for the project would be entirely different. A clear distinction would be drawn between the buildup of the simulator and its use for testing purposes. This distinction might be extended to include the use of different personnel for the two phases. In particular, the simulator would be built and checked out more slowly.

The load-cell system would be modified to insure the impossibility of measuring physically unrealizable forces and moments (that is, not equal and opposite on the two bodies). This could be accomplished by using load cells on only one vehicle; however, some of the simulation realism would be lost. The best method might be to use an overdetermined set, of perhaps nine load cells per vehicle, which would eliminate the noise and bias problems.

Only through the efforts of a large number of people was the success of this simulation possible. It is hoped that some part of the description of the problems, solutions, and experiences will be useful to others who are now involved, or who may become involved, in simulations similar to the one described.

REFERENCES

1 R M DEITERS T NOMURA
Circle test evaluation of a method of compensating hybrid computing error by predicted integral
Simulation vol 7 no 1 pp 33-40 January 1967

2. ANON
Dynamic docking qualification test program
October 1966
North American Aviation Inc Report no SID 66-914

3 R GELMAN
Corrected inputs—a method for improved hybrid simulation
Proc Fall Joint Computer Conference AFIPS 163

4 E G GILBERT
Dynamic-error analysis of digital and combined analog-digital computer systems
Simulation vol 6 no 4 pp 241-257 April 1966

5 W J KARPLUS
Error analysis of hybrid computer systems
Simulation vol 6 no 2 pp 121-136 February 1966

6 R E LANG
Digital integrator for hybrid applications
Instruments and Control Systems vol 40 no 1 pp 103-105 January 1967

7 D L MATLOCK
Pulsed prediction filters applied to digital and hybrid simulation
Simulation vol 6 no 3 pp 153-157 March 1966

8 E E L MITCHEL
The effect of digital compensation for computation delay in a hybrid loop on the roots of the simulated system
Proc Fall Joint Computer Conference AFIPS 1967

9 T MIDRA J IWATA
Effects of digital eecution time in a hybrid computer
Proc Fall Joint Computer Conference AFIPS 1963

10 E L STEWART
Grounds grounds and more grounds
Simulation vol 5 no 2 pp 121-128 August 1965

11 E L STEWART
Noise reduction on interconnect lines
Simulation vol 5 no 3 pp 149-155 September 1965

These pages, 121 - 142 deleted due to circumstances beyond the control of the Fall Joint Computer Conference and AFIPS Press.

Solution of integral equations by hybrid computation

by G. A. BEKEY
and J. C. MALONEY
University of Southern California
Los Angeles, California
and
R. TOMOVIC*
Belgrade University
Belgrade, Yugoslavia

INTRODUCTION

The mathematical description of many problems of engineering interest contains integral equations. Typical of a large class of such problems is the Fredholm integral equation of the second kind,

$$y(x) = f(x) + \lambda \int_a^b K(x,t)\ y(t)\ dt \qquad (1)$$

where f(x) and the kernel K(x,t) are given functions, a and b are constants, λ is a parameter and y(x) is to be found. From a computational point of view, equations of this type may be considered as problems in two dimensions, where one dimension (t) is the dummy variable of integration. For digital computer solution, both variables must be discretized. For analog computer solution, it is possible to perform continuous integration with respect to the variable t for a fixed value of x and perform a scanning process to obtain step changes in the second variable. In either case, the solution is iterative and results in a sequence of functions $\{y_n(x)\}$, $n=1,2,\ldots$ which, under certain conditions, converge to the true solution y(x) as n increases.

It is evident that such a sequential solution, with a two dimensional array K(x,t), may be extremely time consuming for pure digital solution. On the other hand, the scanning and iteration procedures, which require storage of the successive approximation to the solution, do not lend themselves to pure analog computation. Rather, the problems require a hybrid combination of high speed, repetitive integration, memory, and flexible control logic.

The advantage of using such hybrid computational methods for the solution of integral equations was realized quite early. A special purpose computer for solution of integral equations was proposed by Wallman in 1950[1]. Basically, the computer technique consisted of replacing the integration with respect to two independent variables by scanning at two different rates. These original proposals for iterative solution of integral equations were based on the classical or Neumann method.[2] In 1957 M. E. Fisher proposed an iteration technique for high-speed analog computers equipped with a supplementary memory capacity which resulted in considerably faster convergence than the classical technique.[3] However, little practical experience with his method is available due to the complexity of the function storage and playback apparatus.[4] One test of Fisher's method was made using a repetitive analog computer in which the computer operator manually adjusted a set of potentiometers in a special function generator at the end of each iteration cycle.[5]

The purpose of this paper is to examine hybrid computer solution of integral equations, by both the Neumann and Fisher methods.

* This work was initiated during 1964-65, while the author was an NSF Senior Foreign Scholar and Visiting Professor at the University of Southern California, Los Angeles. It was supported in part by the Office of Scientific Research U. S. Air Force under Grant No. AF-AFOSR 1018-67.

The Neumann iteration method

The classical or Neumann iteration procedure for the solution of (1) is specified by

$$y_{n+1}(x) = f(x) + \lambda \int_a^b K(x,t)\, y_n(t)\, dt \qquad (2)$$

with $y_o(t) = 0$. Under conditions discussed in Reference 4, the process converges to a limit $y_\infty(x)$ which is the solution of (1).

From a computer point of view, an integration over the whole range of t must be made for each particular selected value of x. If the range of x is divided arbitrarily into I segments of length Δx, the function is represented by the values of y(x)) at the midpoint of each segment, i.e., $y(x_i)$, $i = 1,2,3,\ldots I$. It is clear that a total of I integrations in the t domain must be made before a single change in $y_n(t)$ is made in equation (2). Such an integration in t for a single value of $x = x_i$ will be called a *minor cycle*. In order to increase the index n, i.e., to derive the next approximating function $y_{n+1}(t)$, one has to complete I minor cycles. This group of minor cycles will be called a *major cycle*. The complete solution theoretically requires an infinite number of major cycles. However, practical experience[4] has demonstrated that accuracies of the order of 1% are attainable in about 20 major cycles using Neumann's method.

It should be noted that digital computer implementation of the strategy defined by (2) requires that the variable t also be discretized and that an appropriate numerical integration formula be used. For example, if Euler or rectangular integration is used, Equation (2) becomes

$$y_{n+1}(x_i) = f(x_i) + \lambda \sum_{j=1}^{J} K(x_i,t_j)\, y_n(t_j)\, \Delta t \qquad (3)$$

$$i = 1,2,\ldots,I$$

where Δt = constant is the integration step size and J is the total number of steps in the interval (b - a). Since t is only a dummy variable, the total range in t must equal the range of x and it is possible to choose the number of steps in t and x to be equal, i.e., let I = J. More sophisticated numerical procedures do not change the need for minor and major integration cycles. Equation (3) requires only the algebraic operations of addition and multiplication and is well suited to digital computation.

For hybrid computer solution each minor cycle may be performed continuously (using analog integration) and the resulting values stored. Assignment of other specific computational functions to the analog or digital portions of a system may significantly affect overall accuracy, as discussed in a later section of the paper. Thus, the generation of the functions f(x) and K(x,t) as well as the multiplication under the integral sign in (2) may be performed either in the analog or digital computer. A flow chart illustrating the programming of Neumann's method is shown in Figure 1. A stopping criterion given in the flow chart is based on reducing the difference between successive approximations to a sufficiently small value.

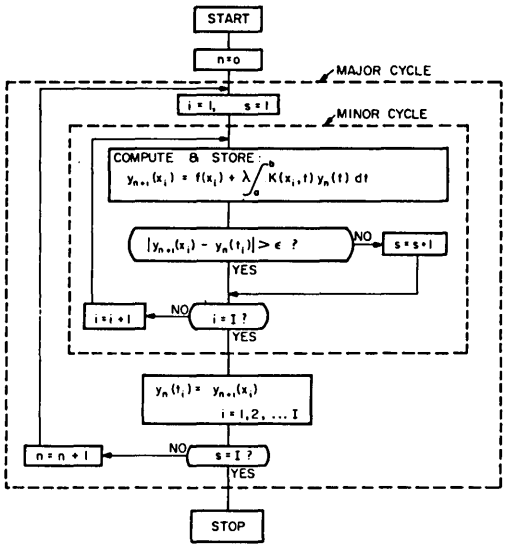


Figure 1—Flow diagram of Neumann's method

The Fisher iteration method

An examination of Equation (2) and Figure 1 reveals that Neumann's method requires the storage of $y_{n+1}(x_i)$ at the end of each minor cycle (i.e. for $i = 1,2,\ldots I$). At the end of I minor cycles the entire vector $[y_n(t)]$ under the integral sign is replaced and a major cycle has been completed.

Fisher's method[3,4] for the solution of the same problem requires that one element of the vector $[y_n(t)]$ be updated at the end of each minor cycle. Consequently, the Fisher version of the digital process of equation (3) becomes

$$y_{n+1}(x_i) = f(x_i) + \lambda \sum_{j=1}^{J} K(x_i,t_j)\, y_{n,i-1}(t_j)\, \Delta t \qquad (4)$$

$$i = 1,2,\ldots I,$$
$$j = 1,2,\ldots,J, \qquad I = J$$

Note that y(t) under the summation sign now carries a double subscript. The idea is to replace at the end of each minor cycle the existing value of $y_n(x_i)$ in the memory with the newly obtained value of $y_{n+1}(x_i)$. The notation (n,i)i=1,2,. . , I implies that during each major cycle the unknown function $y(x_i)$ is gradually adjusted as the index i is increased, not waiting, as in

the Neumann method, until all minor cycles are completed. In other words, Fisher's method is based on using each piece of new information as soon as it is available so that the adjustment from $y_n(x)$ to $y_{n+1}(x)$ proceeds gradually, rather than being performed all at once after I minor cycles.

The hybrid computer version of Fisher's method takes the form

$$y_{n+1}(x_i) = f(x_i)+\lambda \int_a^{x_i} K(x_i,t)\, y_{n+1}(t)dt + \lambda \int_{x_i}^{b} K(x_i,t)\, y_n(t)\, dt \qquad i=1,2,\ldots I \tag{5}$$

The first integral on right hand side of (5) contains the results which have been obtained during the previous minor cycles of the present major cycle. A flow chart showing the hybrid computer implementation of this strategy is shown in Figure 2.

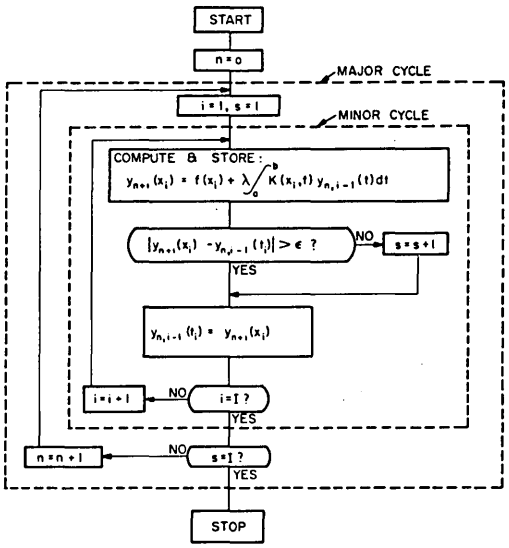


Figure 2—Flow diagram of Fisher's method

Illustrative examples

Fisher has shown[4] that for symmetric kernels his algorithm converges whenever the Neumann algorithm does, and in certain cases also when the classical method fails. (A symmetric kernel is characterized by $K(x,t) = K(t,x)$). Furthermore, using a simple example, Fisher has demonstrated that his method may speed up convergence significantly. In order to obtain practical results concerning this comparison, several problems were solved using a small hybrid computer (IBM 1620 digital computer and Beckman 2132 analog computer).

In order to facilitate the evaluation of the two methods, solutions, $y_n(t)$ were compared with known exact analytical solutions, $z(t)$, by means of a root-sum square criterion, defined by

$$F_n = \sqrt{\sum_{i=1}^{I} [\, z(x_i) - y_n(x_i)\,]^2} \tag{6}$$

Example 1. The following equation was solved:

$$y(x) = \frac{2}{3} + 2\int_0^1 (x-t)\, y(t)\, dt \tag{7}$$

with the initial approximation $y_o(t) = \frac{2}{3}$. The step size was chosen as $\Delta x = 0.1$ and the multiplication was performed on the analog computer. The solutions obtained by the Neumann and Fisher methods are compared in Figure 3 using the criterion F_n defined in (6). The considerably faster convergence of Fisher's method is clearly illustrated.

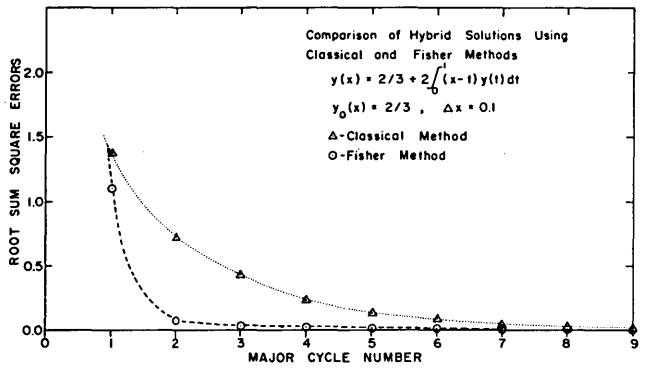


Figure 3—Comparison of hybrid solutions using classical and Fisher methods

Time histories of the kernel $K(x,t) = x-t$ and the functions $y_n(x_i)$ for several major cycles are shown in Figure 4.

Example 2. The following equation was solved.

$$y(x) = 1.5x - \frac{7}{6} + \int_0^1 (x-t)y(t)\, dt \tag{8}$$

To illustrate the effect of choice of initial conditions, the equation was solved once with $y_o(t) = 0$ and once with $y_o(t) = 2/3$. The results are shown in Figure 5 for $\Delta x = 0.01$. It is evident that a poor choice of initial approximation may lengthen the convergence process.

Example 3. Examples 1 and 2 used kernels which

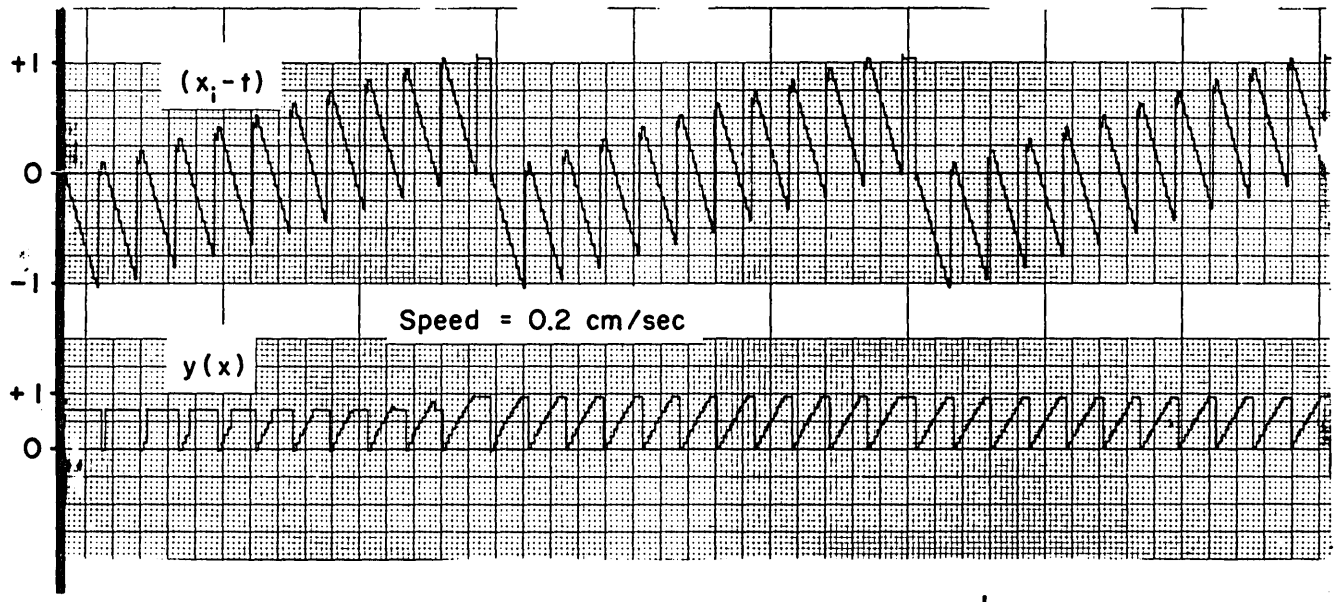


Figure 4—Time history of Fisher's solution

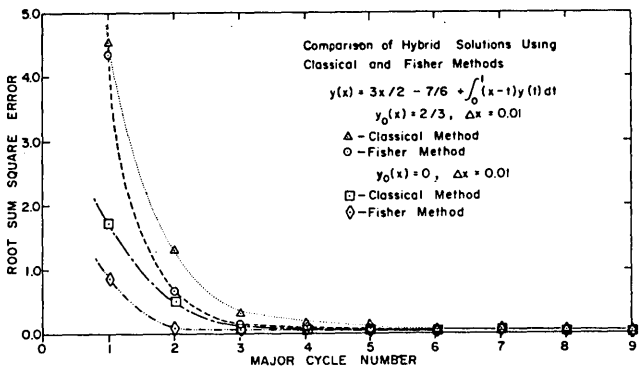


Figure 5—Comparison of hybrid solutions using classical and Fisher methods

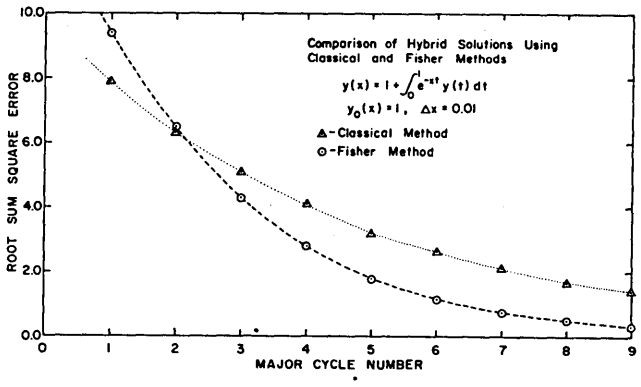


Figure 6—Comparison of hybrid solutions using classical and Fisher methods

could be considered functions of a single variable. Consider now the equation

$$y(x) = 1 + \int_0^1 e^{-xt}\, y(t)\, dt \qquad (9)$$

$$y_0(t) = 0$$

The results are shown in Figure 6. Once again the superiority of Fisher's method is evident.

Discussion of errors

The major errors which enter into the hybrid solutions of integral equations are the following: (a) truncation errors (due to the fact that the functions have been quantized), (b) A/D and D/A conversion errors, (c) other analog computer errors, and (d) phase shifts due to digital execution time.

Truncation errors arise from the quantization of the variables x and t in equation (1). In order to test the

importance of the quantization interval, Example 1 above was solved using 10 intervals ($\Delta x=0.1$) and 100 intervals ($\Delta x=0.01$). A comparison of the root sum squared errors for both interval sizes using the Fisher method is shown in Figure 7. It can be seen that convergence is speeded up by the choice of a smaller interval. After a sufficiently large number of major cycles, there is no apparent advantage to the small interval size. In fact, an oscillation in the final error criterion values ensues with the small Δx, which may be due to a random compounding of roundoff errors from the large number of minor cycles.

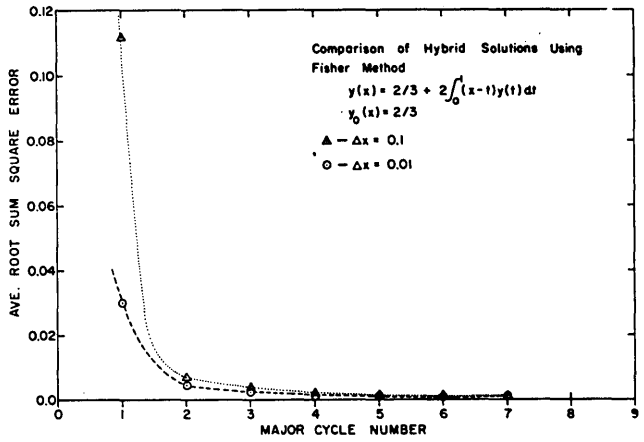


Figure 7—Comparison of hybrid solutions using Fisher method

The effect of quantization also enters into the integration process since the approximating function $y_n(t)$ is reconstructed from the samples $y_n(x_i)$, $i = 1,2,\ldots I$. In the present study this reconstruction was accomplished using simple zero-order holds, i.e.,

$$y_n(t) = y_n(t_i) \text{ for } t_i - \frac{\Delta t}{2} \leq t < t_i + \frac{\Delta t}{2} \qquad (10)$$

It can be shown[4] that with both zero-order and first-order reconstruction the errors are proportional to $(\Delta t)^2$. Second-order interpolation formulas will reduce the error to $0(\Delta t)^3$, but the additional computation required to achieve it may not be justifiable.

Analog-to-digital and digital-to-analog conversion errors were extremely important in the solution of the example problems. However, it should be noted that both the Fisher and Neumann techniques are stable processes as long as the solution to the problem is analytically convergent (see Reference 4). Thus, random errors (which may enter the problem from the converter or multiplier inaccuracies) during any one iteration (major cycle) will be corrected in subsequent iterations. Consequently, random errors delay the convergence process and may also cause a final indeterminacy region in the solution, as evidenced in the oscillation of Figure 7.

Alternate mechanization of kernel generation and multiplication were also investigated. The resulting effects on solution accuracy are clearly a function of the quality and precision of available analog multipliers and function generators, as well as conversion errors. It should be noted however, that analog generation of the kernel K(x,t) has the advantage of producing a program which has a solution time independent of kernel complexity and whose digital portion is universally applicable.

Solution time. While the actual times taken for a given solution clearly depend on the particular digital computer being utilized, comparisons between hybrid and all digital solutions are of interest. A speed up factor of 300 to 1 was obtained by using hybrid computation over digital computation, with comparable final accuracy.

Extension to other types of integral equations

The above discussion has been devoted entirely to equations of the Fredholm type. However, extension of the technique to many other types of equations is possible. Consider, for example, the Volterra equation:

$$y(x) = f(x) + \lambda \int_a^x K(x,t)\, y(t)\, dt \qquad (11)$$

This equation differs from the Fredholm equation in that the upper limit of integration is variable. The algorithms of Figure 1 and 2 are still applicable if the kernel is redefined such that

$$K(x, t) = O \text{ for } t > x \qquad (12)$$

A simple digitally controlled switch (needed to implement (12)) can be used in conjunction with the Fredholm equation program to solve Volterra type equations.

Wallman[1] and Fisher[3] have also indicated possible extension of hybrid techniques to the solution of multidimensional integral equations, integrodifferential equations and certain more general functional equations. Such extensions have yet to be proved in practice.

CONCLUSION

Hybrid computation techniques, involving fast repetitive analog integration and function generation and digital storage and control, are well suited to the solution of integral equations. Hybrid techniques lead to a substantial reduction in solution time when compared to all-digital methods. Further, the examples solved in this study substantiate the faster convergence of Fisher's iteration scheme, when contrasted with the classical

Neumann technique. Extensions to other areas of application appear promising but remain to be tested.

ACKNOWLEDGMENT

The writers express their appreciation to Messrs. L. J. Gaspard and T. Deklyen for their assistance in obtaining a number of the computer solutions discussed in this paper.

REFERENCES

1 H WALLMAN
An electronic integral transform computer and the practical solution of integral equations
J. Franklin Inst. 250:45-61 July 1960

2 F B HILDEBRAND
Methods of applied mathematics
Prentice-Hall Inc Englewood Cliffs N.J. 1952

3 M E FISHER
On the continuous solution of integral equations by an electronic analogue
Proc Cambridge Phil Soc 53:162-173 1957

4 D B McKAY M E FISHER
Analogue computing at ultra-high speed
John Wiley & Sons Inc New York 1962

5 R TOMOVIC N PAREZANOVIC
Solving integral equations on a repetitive differential analyzer
IRE Trans on Elec Computers EC-9 503-506 December 1960

Graphic CRT terminals–characteristics of commercially available equipment

by CARL MACHOVER
Information Displays, Incorporated
Mount Kisco, New York

INTRODUCTION

"Who needs another review of Graphic CRT Terminals when so many good ones have recently been published?"

A reasonable question—

After all, Adams Associates is now offering the "Computer Display Review"[1] — in which they're trying to do for the display field what they have been doing so effectively for the Computer field (with their "Computer Characteristics Quarterly"). The Air Force has just published their "Compendium of Visual Displays"[2]. Harry Poole's recent book, "Fundamentals of Display Systems"[3] admirably meets the author's stated objective ". . . to provide fundamental data and illustrate basic techniques used in the design and development of display systems". Excellent books covering the use of displays in specialized application areas, like James Howard's new book "Electronic Information Displays for Management"[4], are now available. Several survey articles[5,6] have recently appeared. Reasonably complete bibliographies of computer graphics literature have been published, such as the selected bibilography appearing in Ronald L. Wiginton's paper "Graphics and Speech Computer Input and Output for Communications with Humans" inculded in "Computer Graphics/Utility — Production — Art".[7] These examples by no means exhaust the list—but they do emphasize the lead question "Why another one?"

The justification depends on two factors, I think. First, there appears to be a need for user oriented-hardware based information. (There is, to be sure, also a need for user oriented-software based information—but that will have to be the subject of someone else's paper.)

Second, there is also the need, I believe, to define terms. The user encounters words and phrases like "jitter" and "flicker-free" which are used to describe the performance and quality of a display system . . . but the exact meaning of the terms are frequently left undefined. This lack of definitions is not the result of a mass conspiracy by display manufacturers — but instead reflects the absence of standards and definitions in the field. Incidentally, one of the Society for Information Displays goals over the next few years is to establish a concensus on standards and definitions. This paper will attempt to clarify some commonly used display terms, as a guide to user understanding.

Generally, the discussion will be oriented to commercially available equipment.* Table I is a representative list of manufacturers of commercial graphic CRT terminals.

TABLE I

Manufacturers of Commercially Available CRT Graphic Terminals

Bolt, Beranek & Newman, Inc. (BBN)
Bunker-Ramo Corporation (BR)
Control Data Corporation (CDC)
Digital Equipment Corporation (DEC)
Ferranti, Limited
Information Displays, Inc. (IDI)
Information International, Inc. (III)
International Business Machines Corporation (IBM)
International Telephone & Telegraph Corporation (ITT)
Philco-Ford Corporation.
Sanders Associates
Scientific Data Systems, Inc. (SDS)
Stromberg-Carlson Corporation (SC)
Systems Engineering Laboratories, Inc. (SEL)
Tasker Instruments Corporation
UNIVAC

*This is a good place to start defining terms. By "commercially available" I mean relatively standard products regularly offered for sale — and meant to be used in an industrial rather than military environment. Admittedly, this is arbitrary (but, I hope, not capricious) and eliminates from consideration the numerous militarized command and control systems — and the fine display systems developed in University and Industrial Laboratories.

Photographs of representative terminals are included in Figure 1.

Section 2 of this paper briefly reviews the constituent elements of a typical Graphic CRT terminal.

Prices of these terminals range from approximately $20,000 to $280,000. Generally, as discussed in an earlier article,[8] price and performance are related. However, because there are many factors that affect performance, and the importance of each factor depends upon the application, the cost-performance relationship is not a simple one. Section 3 of this paper is a discussion of various factors which affect performance.

Block diagram

A typical block diagram for a Graphic CRT Terminal is shown in Figure 2. Generally, the terminal consists of:

A *Direct View Cathode Ray Tube* with associated analog deflection and video (intensification) circuitry, plus . . .

A *Display Generator* which contains several types of function generators to produce graphic elements, plus (occasionally) . . .

A *Storage Element,* plus . . . . . .

An *Interface* between the computer and terminal, plus . . .

A variety of *Input Devices* from the display to the computer.

Cathode ray tube

The CRT may be deflected electrostatically, or electromagnetically or by a combination of both. Commonly used configurations are illustrated in Figure 3. Table II lists the deflection system used by various manufacturers.

Most commercially available Graphic CRT Consoles use random positioning to produce a picture. That is, the beam is moved to the desired location on the screen and then controlled to produce the graphic element (line, character, etc.). This is in contrast to the raster scan used in a TV set (and in most alphanumeric CRT inquiry units). Philco is one of the few suppliers of a raster scan graphic CRT terminal. Figure 4 illustrates the difference between random positioning and raster scan.

Tubes are available with a variety of conventional phosphors offering a choice of color as well as persistence. Bolt Beraneck and Newman Teleputer System uses a CRT with storage screen.

So that an optically projected image can be combined with the electronically generated image, CRT's in terminals supplied by Bunker-Ramo and Stromberg-Carlson have a window through which a picture

TABLE II

Deflection Systems Used by Various Manufacturers

Manufacturer	Electrostatic	Magnetic	Dual Deflection: Electrostatic + Magnetic	Dual Deflection: Magnetic + Magnetic
BBN	x			
BR				x
CDC	x			
DEC		x		
Ferranti		x		
SC			x	
IBM				x
IDI			x	
III				x
ITT		x		
Philco				x
Sanders				x
SDS		x		
SEL			x	
Tasker				x
UNIVAC			x	

can be projected onto the screen. This is illustrated in Figure 5.

Display generator

The Display Generator contains the circuitry which interprets the computer digital data word and translates it into analog signals to generate the graphic elements on the CRT. The Display Generator is essentially just a digital-to-analog converter (D/A). Typically, the Display Generator will include such function generators as a character generator, a vector (line) generator, a circle generator, position generators (conventional D/A converters) and a dot generator.

All of the graphic elements produced by the various function generators could also be produced by appropriate computer software. For example, letters could be programmed as a series of dots, or short line segments. However, each dot or line segment requires a separate digital instruction from the computer and this tends to increase the computer programming. Therefore, although there are under development several interesting and commercially promising graphic terminals using this technique, all currently available commercial terminals, except the BBN Teleputer units, use special purpose function generators.

Instead of using a separate character function generator in the Display Generator, the Stromberg-Carlson console uses a special CRT which shapes the cross-section of the electron beam into the desired character. This is accomplished by inserting a stencil mask in the

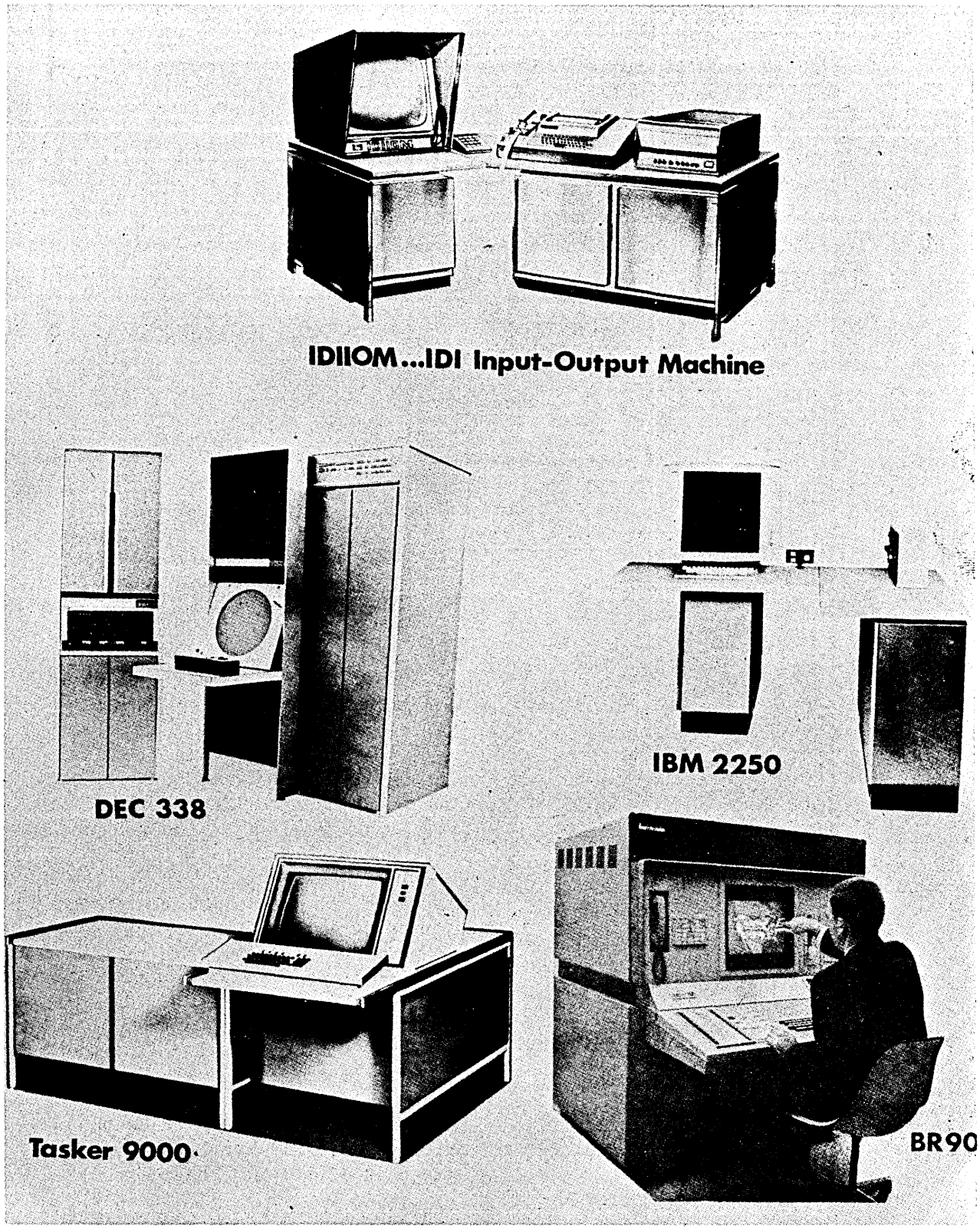


Figure 1—Typical graphic CRT consoles

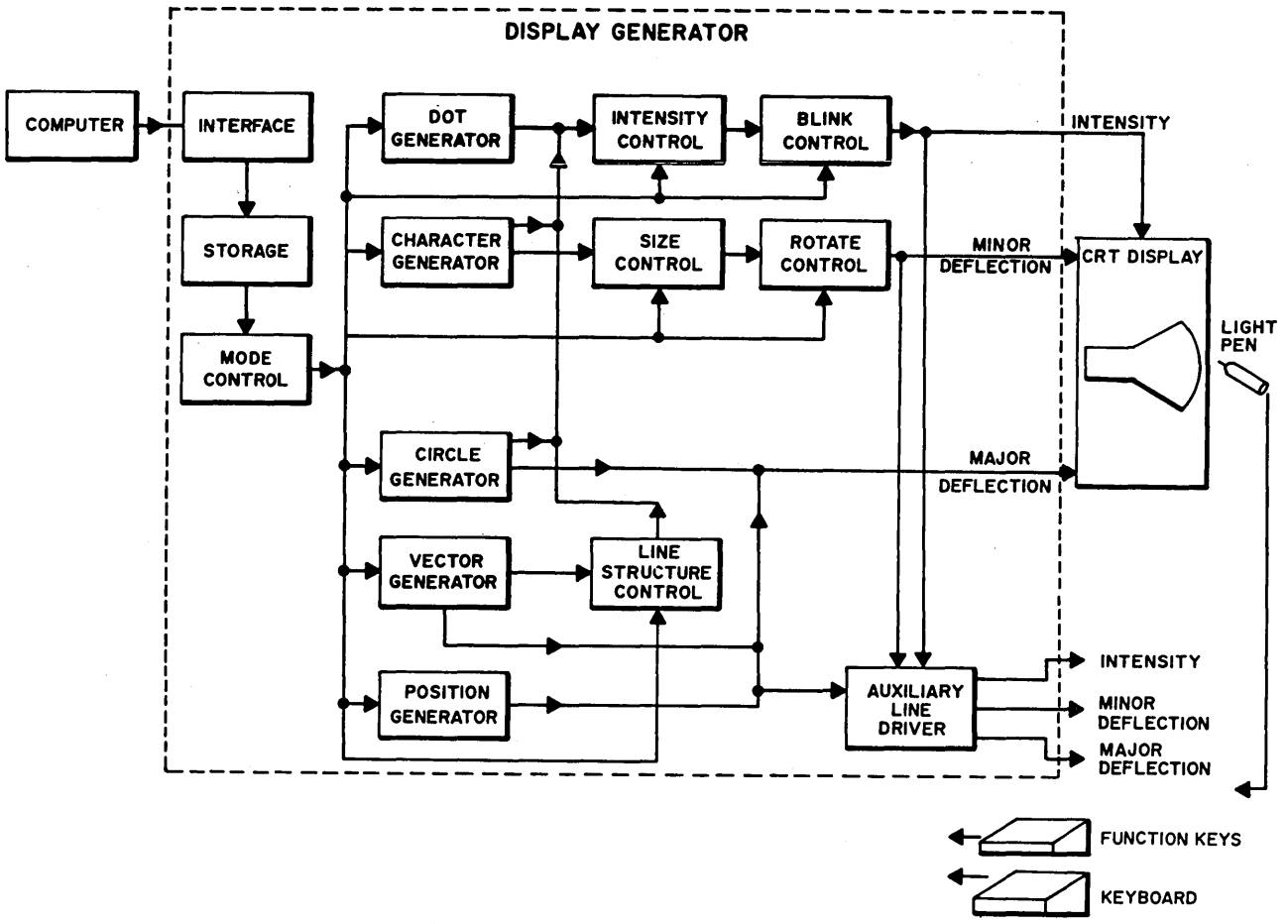


Figure 2—Graphic CRT display

CRT. The beam is extruded through the appropriate aperture and the beam cross-section assumes the shape of the selected character.

Terminals often contain several other modules which modify the outputs of the function generators. In IDI terminals, for example, a digital size control is usually associated with the character generator. By using one or two digital bits, two or four character sizes can be programmed. A line structure control is sometimes included. This permits programming a line to be dotted, dashed, or dash-dotted as well as solid. While this could also be done with software, the hardware alternate allows the control to be done with one or two digital bits, without increasing line drawing time. Hardware is often included so that the intensities of graphic symbols can be digitally controlled. Any of the graphic elements can also be made to blink, that is, turned off and on at low frequency (about 5 cps).

At the input to the Display Generator is a Mode Control. The Mode Control decodes the computer data word and activates the appropriate function generators and modifiers. In most terminals, several other hardware functions are assigned to the Mode Control. For example, characters can be arrayed in typewriter fashion, as on this page, with the Mode Control, Spacing can be automatically adjusted to character size. When the end of the line is reached, the line can be automatically reset to the left hand margin (or to a programmed margin) and then advanced to start the next line of characters. Characters can be rotated CCW by 90° and written vertically starting at the bottom of the screen; or the characters can be superscripted and subscripted. Logic for connecting (or stringing) line segments can be included. Once the first line is established, only the successive end points of the line segments need be outputted to the Display Generator. Special modes for curve or graphic drawings can be incorporated. The Mode Control may include address and index registers to facilitate programming.

The design of the Mode Control represents the manufacturer's trade-off decision betwen computer software and display hardware.

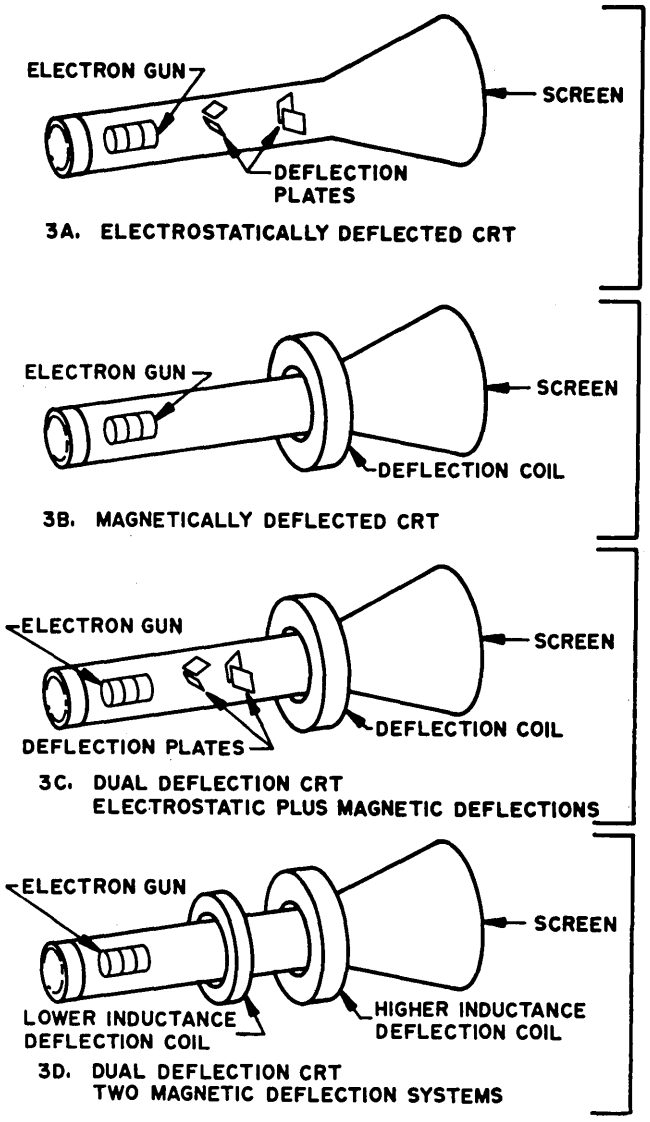


Figure 3—Commonly used deflection systems

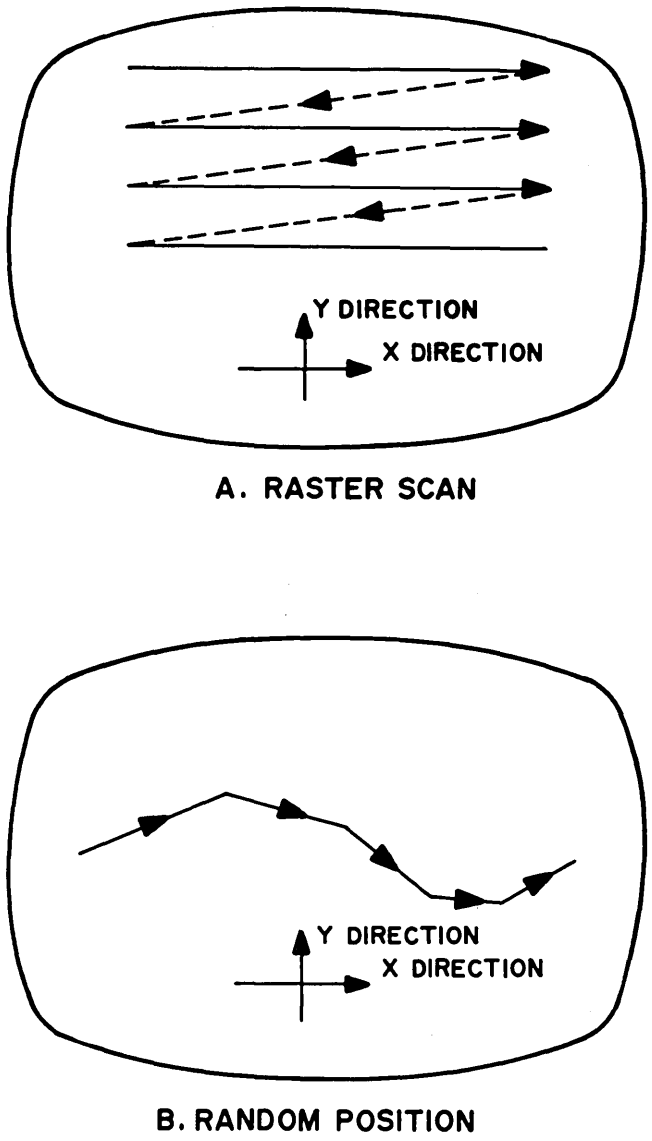


Figure 4—Basic deflection schemes

Storage

In most CRT graphic terminals, the image must be repeated (refreshed) in order to appear flicker-free* to the user. If a terminal does not contain a storage device, the computer must continually refresh the display.

Many modern computers have a memory configuration which can be used to refresh the display without interrupting other computations. Where this is not possible, a buffer memory is available within a display. Commercial terminals use core memories, delay lines, and drums. With the availability of low cost, high-speed, general purpose, digital computers, it becomes feasible to consider including a digital computer in the CRT graphic terminal. BR, DEC, and IDI offer terminals in which the digital computer is an integral part of the display and provides functions of storage, plus some of the hardware mode control features.

As stated earlier, BBN terminals use a storage CRT so that the image need be written only once and the local memory is not required. However, these storage tubes cannot be easily used in systems in which the operator wants to use a light pen to input graphical data to the computer.

*"Flicker-free" is a term which will be discussed in Section 3.

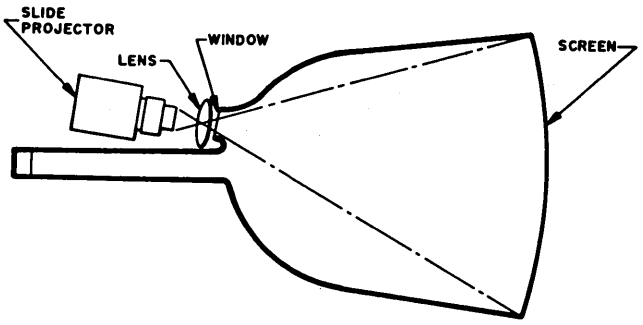


Figure 5—Using a ported (window) CRT to combine projected

Interface

Because CRT graphic terminals are available from manufacturers who do not make the computer with which the display is to operate, an interface is usually required to convert the computer output into a form suitable for the terminal. The conversion may include reorganization of the data words, conversion of logic levels, and generation of appropriate communication signals.

Input devices

The discussion thus far has been about a one-way device; information is received from a computer, converted in the Display Generator, and presented as a graphic image on a CRT. However, one of the basic reasons for the increasing acceptance of CRT graphic terminals is the availability of an operator channel from the display back to the computer. The operator can converse with the computer on-line* and in real-time* using input devices such as a light pen, joy stick, track ball, Rand tablet, and function keys.

Factors which affect performance

As can be inferred from the foregoing discussion, a graphic CRT terminal is a conglomerate of devices, each of which has a range of characteristics which can affect the performance and useability of the terminal. For convenience, the various factors which contribute to the effectiveness of the terminal can be grouped into three categories:

1. Those which affect the *data content* . . . that is, how much information can be displayed simultaneously without flickering objectionally and with the graphic symbols large enough to be easily read; or, when small, to be easily distinguishable.
2. Those which affect the *quality* of the display . . . that is, how the display looks aesthetically to the observer, and
3. Those which affect the *ease of use* . . . both from a human factors standpoint and from a systems programming standpoint.

There is, of course, some "spill-over" among categories, but the categories provide a convenient frame of reference for the discussion which follows.

Data content

The amount of data which can be displayed simultaneously, without appearing to flicker cannot be determined until one explores the concept of "without appearing to flicker."

"Without appearing to flicker" or alternately, "flicker-free" is not a factor to which a single number can be assigned (as most manufacturer's are prone to do). Perceptible flicker varies with individuals and has been found to be a function of such factors as the individual's age, the color of the light, whether the individual is looking directly at the object or looking out of the corner of his eye, the brightness, the brightness variation, the variation between light and dark, and the size of the flickering object. Typical studies of flicker[10] use phrases like "the average observer" or "90% of the observers" to bound statements about observable flicker. H. Poole[2] presents a curve, reproduced as Figure 6, which relates critical frequency (frequency below which flicker is observed) to brightness. However, the data in Figure 6 make no allowance for the persistence of the CRT phosphor.

The typical CRT non-storage phosphor used in commercially available consoles retains an image for times* ranging from about 40 USEC to 0.6 seconds, as summarized in Table III. For these typical phosphors, J. Bryden[10] has experimentally determined (for a specified test condition) the "lowest refresh rate which will give freedom from flicker for 90% of the observers," and this experimental value is tabulated in Column 6 of Table III. Note that although Figure 6 indicates a critical frequency of about 50 cps at a

*"On-line" and "real-time" are fairly abused terms in the computer world, and are used here without being rigorously defined. The sense is to describe J.G.R. Licklider's ". . . 'on-line people,' which is to say, people who are interacting directly with information and with information processors."[9]

*Phosphors are categorized by the length of time required for the image to decay to 10% of its initial value. Phosphor classifications are summarized below.

Time Required to Decay to 10% of Initial Brightness	Phosphor Classification
Less than 1 USEC	Very Short (VS)
1 USEC to 10 USEC	Short (S)
10 USEC to 1 MS	Medium Short (MS)
1 MS. to 0.1 SEC	Medium (M)
0.1 SEC to 1 SEC	Long (L)
Longer than 1 SEC	Very Long (VL)

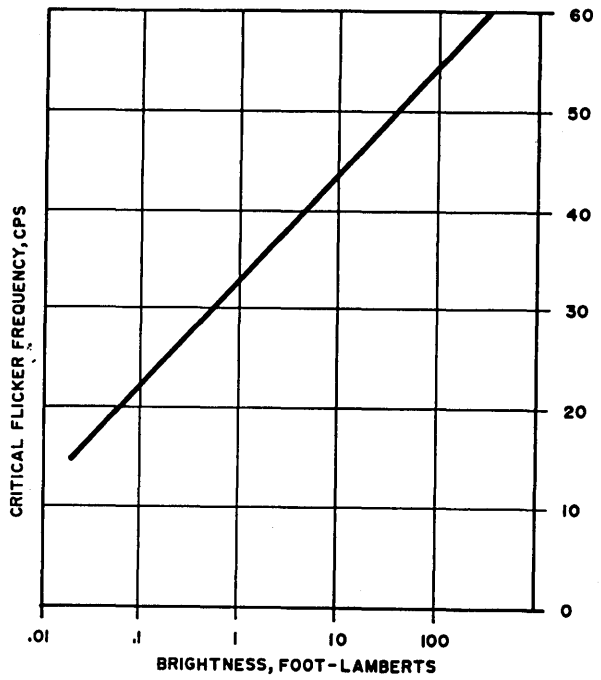


Figure 6—Critical flicker frequency as a function of brightness

brightness of 50 foot-lamberts, the data in Table III show that 90% of the observers will not see flicker at frequencies greater than 21 cps to 38 cps, depending on phosphor. J. Bryden presents another table (Table 4.1) which lists "Refresh Rate (cps) for Flicker Threshold of *Average* Person at 50 foot-lamberts". These values, listed in Column (7) of Table III range from 17.5 cps to 33.5 cps depending on phosphor.

The purpose of detailing the varied results is to emphasize that a simple statement of "flicker-free" presentation is an inadequate user (or manufacturer) specification. (Of course, if the console uses a storage CRT, the image need not be refreshed until intentionally erased, or until decay after relatively long time measured in minutes; and the question of a flicker-free presentation is not meaningful.)

a) *Frame Rate*

The basic problem, then, is deciding how many times per second the image needs be repeated (or refreshed) so that it does not appear to flicker to the observer. Having decided this flicker-free rate, the time available to write a complete frame of data is simply the reciprocal of the rate . . . the faster the refresh rate, the shorter the time available per frame.

Why not always use a longer persistent phosphor, thereby reducing frame rate and increasing time per frame? For several reasons: (1) Shorter persistent phosphors, such as the P31 (medium-short) tend to be more visually efficient and look brighter to the user. This is illustrated by Column (5) of Table III which lists J. Bryden's test results showing the brightness for several phosphors under identical test conditions. (2) Long persistent phosphors, like the P19 and P28, are more easily burned than the medium-short persistence P4 and P31. (3) If the terminal is displaying a rapidly moving graphic element, the long-persistent phosphors tend to smear the image. (4) Brightness is a direct function of frame rate.

In order to provide a standard reference in the balance of this paper, *all frame-time dependent values and references to flicker-free presentations in the following discussions are based on a 40 cycle frame rate (or 25 MS frame interval).* Keep in mind, however, that the data content can be increased in direct proportion to frame interval. For example, if a flicker-free data content of 250 points is quoted, it is understood that the calculations are based on 25 MS frame interval.

If, for example, in a particular system, with a particular phosphor, a frame interval of 50 MS (frame rate = 20/second) were acceptable, the number of points could be increased to 500.

b) *Deflection Amplifier Response*

A primary parameter of a CRT display is the speed with which the beam can be positioned. In random positioned systems, the beam can be moved anywhere on the screen in times ranging from 3 USEC to 100 USEC. A complete specification of deflection time should include a statement of the accuracy to which the beam has settled within the quoted time. For example, the quoted random positioning time of IDI systems is based on settling to within $\pm 0.1\%$ of final value. Until recently, electrostatically deflected CRT's tended to exhibit the fastest random positioning times. The new electromagnetically deflected Tasker console, however, is reported to have random positioning times as fast as commercially available electrostatic units.

The number of random dots which can be displayed, flicker-free, ranges from 250 to 8300.

If the data can be properly formatted, it may be possible to organize the information so that full screen random positioning from dot to dot is not required. Under these circumstances, the small angle (incremental) positioning time of the deflection amplifier is critical. In commercially available equipment, this ranges from 1 USEC to 10 USEC. Therefore, the number of incremental dots which can be displayed flicker-free ranges from 2500 to 25000.

Note that these factors are based on the assumption that the system uses random positioning. There are systems, however, which use a raster scan similar to that used in a conventional television set. Such systems require complete formatting of the data. However, by doing so, as many as 1,000,000 dots can be displayed flicker-free, compared to a maximum of 25,000 dots in a random positioning system.

As illustrated in Figure 3C and 3D, and Table II, many consoles employ dual deflection. This second deflection channel, typically wide bandwidth (DC to 50 MC) and small angle deflection, is used for character writing.

c) *Character Writing Time*

Character generators available in commercially available terminals write a character in times ranging from 2 USEC to 100 USEC. To these times must be added the positioning time (ranging from 3 USEC to 100 USEC random or 1 USEC to 10 USEC, small angle). Therefore the number of random characters that can be displayed flicker-free ranges from 125 to 5000, and the number of formatted character (text) that can be displayed flicker-free ranges from 220 to 8300. As a comparison, a typical double spaced typewritten page contains about 2500 characters.

d) *Line Drawing Time*

Two types of vector (line) generators are offered in commercially available equipments. One type requires a fixed time to draw a line regardless of line length, while the other requires a time proportional to line length. Typical fixed time vector generators require from 30 to 150 USEC to draw lines up to full screen size, while the proportional vector generators have line drawing speeds ranging from 0.5 USEC per inch to 150 USEC per inch. This means that the fixed time vector generators can draw between 160 and 830 flicker-free line segments per frame. Depending on CRT screen size, this can represent up to 16,000 inches of line. Proportional vector generators can draw from 160 to 50,000 inches of flicker-free line per frame.

e) *Circle Drawing Time*

Hardware circle generators available with some terminals can generally draw any size circle in from 100 USEC to 300 USEC. This means that from 83 circles to 250 circles can be displayed flicker-free.

f) *Logic Time*

In addition to the actual time required to deflect a beam and write the various functions, the logic in the Display Generator, the logic of the computer, and the memory cycle time will affect the amount of data which can be displayed. For example, the word organization of the terminal may require that the generation of each graphic element be controlled by several data words. Elapsed time from the display's request for a data word and the terminal's set-up is in the order of 1 to 4 USEC. If 1,000 data words were required per frame (a typical value) and between 1 and 4 USEC were consumed in data transfer and logic set-up, 4% to 16% of the frame time would be consumed and the data content would be reduced by that amount.

g) *Resolution*

Resolution determines such things as the smallest readable characer that can be displayed, and the minimum spacing that can be discerned between lines. Basically, CRT beam spot size determines resolution. In commercially available terminals, the nominal spot will range from .01″ diameter to .03″ diameter.* However, spot size might vary by a factor of 3:1 (on the same terminal) because of beam intensity and spot position on the screen (better in the center, poorer at the edges). Some terminals use dynamic focussing techniques to keep the spot size relatively constant over the display area.

Without being rigorous about defining resolution and spot size, one can observe that the resolution of commercially available terminals is such that the number of readable characters per inch ranges from about 3 to 11. For comparison, a Pica Typewriter spaces characters 10 per inch, and an Elite Typewriter spaces characters 12 per inch.

Addressability is sometimes confused with resolution. Addressability is a statement of how many digital positions can be programmed (but not necessarily distinguished) along each axis. Typical terminals offer 9 bit (512) or 10 bit (1024) addressable locations.

h) *Screen Size*

Screen size affects data content primarily from the standpoint of resolution. CRT's used in commercial terminals generally range from 16″ round to 24″ round,* with available display areas of from 10″ × 10″ to 16″ × 16″. Therefore, the number of readable characters per line can range from 30 (based upon a 10″ line of 3 characters/inch) to 176 (based upon a 16″ lnie of 11 characters/inch).

*Determination of spot size, and the correlation between spot size and resolution is another area requiring the attention of a standards committee. J. Bryden's paper includes an informative discussion of his measurement techniques.

*Except the BBN terminal which uses a 5" CRT.

i) *Overlays*

Static information can be superimposed on the beam written data by projecting pictures (slides) through an optical port in the CRT. Typical systems can select from among 25-150 slides.

Quality

Several factors affect the image quality. Some of these factors may also affect data content, as indicated in the following discussion:

a) *Accuracy*—Accuracy describes how the programmed position of the beam corresponds to some eternal reference. For example, if a grid were scribed in the face of the CRT and the beam were programmed with a digital instruction which should cause the beam to fall at a grid line intersection, the variance between the beam position and the intersection is the accuracy. Commercially available systems have accuracies ranging from 1% to 5% of full scale. (Compared, for example, to a reasonably well adjusted frame TV set which ranges from 10% to 15%.) Generally, pictures drawn with this accuracy are quite acceptable to the observer *provided* there is no attempt to superimpose the electronic image with a mechanical reference. The presence of the mechanical reference will emphasize the inaccuracy of the display.

b) *Short-time Stability*—Short-time stability of the image will affert the observer's reaction to it. Small movement of the graphic element, called jitter, can be quite objectionable when it occurs at low frequencies (less than 10 cps). Jitter results mainly from a beat between the frame face and the power line frequency (or submultiple frequency). In an adequately shielded terminal, the jitter is about 0.5 to 1 spot—but even this value can be disturbing to the user.

Two methods are used to reduce or eliminate the apparent jitter. One is to maintain the frame rate at such a value that the beat frequency is relatively high . . . typically 20 cps. Although jitter still may be present, the graphic element is moving so fast that, to the observer, the line or dot simply thickens a bit. Hence, the resolution is affected, but the image appears stationary. This technique is especially successful when used with longer persistent phosphors.

Alternately, since the jitter most frequently comes from stray magnetic fields, the display frame rate can be locked to the line frequency and the jitter essentially eliminated.

c) *Repeatability*—When the beam is programmed to the same location from various places on the screen, the successive dots will probably not be superimposed. The spread, called repeatability, may range from 1 spot size to 10 spot sizes. In commercially available equipment this effect may be particularly disturbing when various line segments are programmed to start from the same point, but, because of repeatability, they do not.

d) *Brightness and Contrast*—If the display is to be used in a normally lighted room, it is important that the presentation be bright or have a high contrast ratio. Typical terminals produce 20 foot-lambert to 50-foot lambert presentations. Medium-short persistent phosphors, such as the P4 and P31 do produce bright, easily read displays, but these phosphors require relatively high frame rates to reduce flicker. Long persistent phosphors, such as P19 and the P28, reduce the frame rate requirements at the expense of brightness. Therefore, displays using long persistent phosphors may require subdued room illumination. Contrast can be enchanced with neutral density filters. Although these filters reduce total brightness, they do increase and enhance the readability of the display.

e) *Phosphor Color*—Phosphors are available which produce white, green, yellow, blue, and red outputs (and shades in-between). The medium-short persistent phosphors are generally in the white, green and blue range; while the long persistent phosphors are in the orange, yellow range. See Table III.*

f) *Graphic Symbol Construction*—Graphtic elements can be constructen in a variety of ways[11] . . . some of which enhance the quality of the display and others which tend to detract from it. For example, characters formed from a 5×7 dot format may be readable but not aesthetically satisfying. Other graphic elements constructed from a series of dots may be readable, but not pleasing.

Stroke characters usually produce acceptable quality. The beam forming and monoscope* techniques permit a wide range of character formats, with few limits on character style. Higher resolution dot formats, typically 16×16, are also capable of producing excellent quality symbols.

Ease of use

There are two categories discussed in this section; those which are based on human factor considerations and those which are based on programming considerations.

a) *Human Factors*—The plane of the CRT screen

*The monoscope produces character video by election beam scanning of a target on which characters have been drawn in ink. Typical commercial monoscopes, such as the Raytheon Symbolray, use targets with 64 or 96 symbols.

*Table III is shown on page 159

ranges from vertical to approximately 45° from the vertical. Generally, this is fixed although a CDC unit features tiltable display screen (essentially continuously variable between vertical and horizontal).

A variety of light pen configurations are available ranging from a simple penholder type to a gun type. Some pens are relatively heavy while others are light weight. Some use a very flexible cable and others use a rather stiff cable or coil cord. Aiming circles are provided with some light pens so that the operator knows where the sensitive area of the light pen is pointed. Activating switches for the light pen include mechanical shutters on the pen, electrical switches on the pen, knee switches and foot pedal switches.

Other operator input devices are available on various consoles. Alphanumeric keyboards and function keys are used. Some function keys use plastic overlays for additional coding. Track balls and joy sticks are preferred by some users. The Rand Tablet, which provides an easy method for graphic input, is available as an accessory in several systems.

Operator controls on commercially available terminals range from only an on-off power switch, to a group of manual adjustments for various display parameters.

Servicing facilities incorporated in terminals range from logic card extender to elaborate maintenance panels, which include register lights and test pattern generators.

Terminal packaging ranges from multiple cabinet configurations, with the display console separated from the display generator, to single cabinet integrated units which occupy 10-15 square feet of floor space, and are 4-5 feet high.

b) *Systems Programming*—The display command structure influences system programming. Two common types of command structures are shown in Figure 7. In one approach, illustrated by Figure 7A, each data word is completely self-contained and has a mode instruction and all other information required to define a graphic element. In contrast, the word organization currently favored (Figure 7B) establishes a mode of operation with one word and then uses a series of succeeding data words to program identical kinds of graphic elements. Figure 7B also illustrates a word organization which includes computer-type instructions such as JUMP and JUMP AND SAVE.

Some graphic terminals such as the CDC, DEC, IBM, and IDI are designed so that more than one display console can be driven from a common Display Generator. These other consoles may be slaved (and have identical information) or they may have

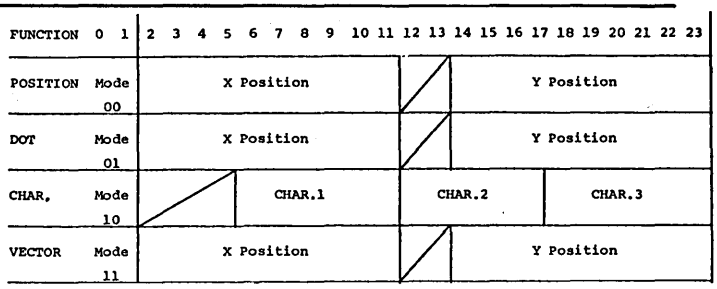


FIGURE 7A

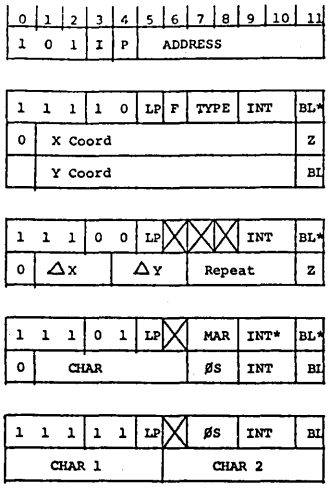
TYPICAL WORD ORGANIZATION

(MODE CONTROL CODE IN EACH WORD)

FIGURE 7B

TYPICAL WORD ORGANIZATION

(CONTROL WORDS AND DATA WORDS)



LP = disable light pen
BL = enable blink
Z = blank
F = frame sync
INT = 00 - normal
01 - dim
10 - bright
11 - off
TYPE = 00 - Position
01 - Position, Write Dot
10 - dash
11 - solid

X, Y = 001 +1
010 +2
111 -1
110 -2
Others - no increment
ØS = 00 - normal, no offset
01 - small, superscript
10 - small, subscript
11 - small, no offset
MAR = 00 - NØP
01 - set margin
10 - return to margin
11 - return to margin and line feed

*These bits do not actually control the Display, but are included to provide programming mode compatibility.

Figure 7—Typical word organization

different information. Such displays may be photographed or used to produce wall size pictures or immediate hard copy.

SUMMARY

This paper has discussed the characteristics of commerically available terminals from an equipment viewpoint — not from an applications viewpoint. One can list a number of current and potential applications for CRT graphic terminals, but data which describe terminal requirements in terms of these applications are

scarce. For example, the line drawing needs of a terminal used by civil engineers for cut-and-fill analysis may be quite different from those of an engineer using the terminal to design integrated circuit masks. Adams Associates, in their "Computer Display Review" formulated several typical presentations including a schematic diagram, floor plan, and weather map — and, using terminal manufacturer's supplied performance specifications, analyzed how long each terminal would take to write the display.

Generally, though, the terminal user considers his data (applications) content requirements proprietary, and seldom publishes his findings. We can expect, however, that over the next few years, many more "How Application Factors Determine CRT Terminal Specifications" papers to appear.

REFERENCES

1 *Computer display review*
Adams Associates Inc 128 Great Road Bedford Massachusetts

2 *Compendium of visual displays*
(RADC Compendium 67-1, Rome Air Development Center Research and Technology Division Air Force Systems Command USAF Griffiss Air Force Base New York March 1967) 2nd revision

3 H H POOLE
Fundamentals of display systems
Sparton Books Washington D C 1966

4 J H HOWARD
Electronic information displays for management
American Data Processing Inc. Detroit Michigan 1966

5 L C HOBBS
Display applications and technology
Proceedings of the IEEE vol 54 no 12 December 1966

6 D B PARKER
Solving problems in graphical dialog
IEEE Computer Group News vol 1 no 2 September 1966

7 R L WIGINGTON
Graphics and speech computer input and output for communication with humans
Computer Graphics Utility—Production—Art
Thompson Book Co. Washington D C 1967 pp 92-96

8 C MACHOVER
Family of computer-controlled CRT graphic displays
Information Displays Inc July/August 1966 pp 43, 46

9 J C R LICKLIDER
Graphic input—a survey of techniques
Computer Graphics Utility—Production—Art
Thompson Book Co Washington D C 1967 p 43

10 J E BRYDEN
Some notes on measuring performance of phosphors in CRT displays
Seventh National Symposium on Information Display, Technical Session Proceedings Society for Information Display 1966

11 C MACHOVER
Converting data to human interpretable form
Data Systems Design vol 1 no 9 September 1964

TYPE (1)	EMISSION COLOR FLUORESCENCE (2)	EMISSION COLOR PHOSPHORESCENCE (3)	PERSISTENCE (4) (SEC)	LUMINOUS OUTPUT FOOT-LAMBERTS (5)	FLICKER FREE RATES (CPS) AT 50 FT. - LAMBERTS Average Person (6)	90% of Observers (7)
P4	White	White	6×10^{-5}	5.2	33.5	38.
P7	Blue	Yellowish Green	Blue 5×10^{-5} Yellow $v \times 10^{-1}$	5.2 3.0	29.8	33.
P19	Orange	Orange	2×10^{-1}	3.0	17.5	21.
P28	Yellow Green	Yellow Green	6×10^{-1}	5.2	31.4	38.
P31	Green	Green	4×10^{-6}	12.0	32.5	36.

NOTE: Column (2) Fluorescence is the light emitted by the phosphor during the period of electron beam excitation.
Column (3) Phosphorescence is the light emitted by the phosphor after the electron beam excitation is removed.
Column (4) Time for initial output to decay to 10% of initial value.
Column (5) From Table 3.1 of Reference 10.
Column (6) From Table 4.1 of Reference 10.
Column (7) From Figure 4.4 of Reference 10.

TABLE III

TYPICAL PHOSPHORS USED IN COMMERCIALLY AVAILABLE GRAPHIC CRT TERMINALS.....
A SUMMARY OF CHARACTERISTICS.

How do we stand on the big board?

by MURRAY L. KESSELMAN
Rome Air Development Center
Griffiss AFB, New York

INTRODUCTION

When is a display a large scale display? There are no hard and fast rules for answering this question. An arbitrary, but convenient starting point is to say that anything larger than 30 inches is considered large scale because 30 inches is the practical limit on cathode ray tube (CRT) size. Why should we want a large scale display? The most obvious reason is that many people have a need to view the same display surface and as the audience grows larger, so must the size of the display.

To illustrate this, Table I provides estimates of the audience area available for views of a given display size. The actual number of viewers will depend on the space allocated to each viewer.

TABLE I

SCREEN SIZE VS AUDIENCE AREA

WIDTH OF SCREEN	AUDIENCE AREA (sq ft)
40 in	120
60 in	340
7 ft	654
8 ft	848
9 ft	1078
10 ft	1338

This table is based on a rule of thumb that no viewer should be closer than two times the maximum screen dimension nor further than six times. The maximum viewing angle of any observer to the plane of the screen is considered as being limited to 60° in making these calculations.

As we have seen, the size of the display is determined by the number of viewers. You may then ask the question "why not provide each viewer with an individual display?". This is plausible under certain conditions, but large scale displays have unique characteristics that make them desirable for many applications. Some of these are:

1. In many cases, it is advantageous for an operator to have information concerning the overall environment within which he is functioning. He is better able to anticipate future problem areas and he is in a position to assume the functions of associated operators when he has knowledge of the overall system status.
2. In presenting available information to a group of individuals by means of a common display, there is some assurance that all persons are reacting to a common data base. This is a critical consideration in a rapidly changing environment where many people are pursuing interrelated tasks toward a common goal. When operators are "buried" within the confines of a console, the particular information being reacted to might be out of date or differ in some respects.
3. In an environment where a group of people require the same or very closely related information, a large display may be more economical of money and space. The addition of individual consoles may provide nothing but a duplication of data that are already available in a form suitable to the accomplishment of a particular mission.
4. The physical configuration of a particular work space may indicate one centrally accessible source of information rather than a number of independent ones. A particular environment may have size limitations that only a large display can adequately serve. An unused wall may be the only means available for the presentation of required information.
5. One display may present less of a maintenance problem than would a large number of consoles that require periodic servicing. This could result in a saving of manpower and of downtime with regard to overall display capability.

6. When one individual can control the composition of a display, he can "force" the attention of all individuals concerned to one problem area. The capability to direct group activities in a required direction ensures that the lower echelons of a unit have the same reference point as the person making the ultimate decisions. All observers are using the same data base as indicated previously.

7. A large display can be viewed from a greater number of locations than can an individual console. The factor permits a degree of mobility and flexibility among interested personnel not afforded to individual console operators.

Use of large scale display

As can be expected, the uses of large scale displays are varied and manifold. It is beyond the realm of this paper to discuss this subject in any detail. The following is a partial list of present and future applications of large scale computer driven displays:

1. Briefing
2. Corporate Planning
3. Simulation
4. Business Gaming
5. Situation Monitoring
6. Traffic Handling
7. Training
8. Education
9. Marketing Analysis
10. Product Planning

Current state of the art

There are a limited number of available off-the-shelf approaches to the generation of automated large scale displays. The current technology includes rapid process film systems, scribe systems, light valves, and projection CRT's. A brief discussion of each technique follows:

1. Film systems

Present photo-recording equipments consist of a cathode ray tube (CRT) which provides a source image, a light sensitive medium on which the CRT image is recorded, and a processor which produces a positive transparency suitable for projection (Figure 1).

Special cathode ray tubes are used to convert electrical signals to alphanumeric and vector images suitable for photographic recording. These are generally five or seven inch flat face, high resolution CRT's with electromagnetic deflection and focus, and use P-11 phosphor as the light emitter. The cathode ray tube image is focused onto a film by means of a special camera. To form a multicolor image, three or four cameras are operated in parallel with each camera exposing a different portion of the film. During projection, each film area is projected with a different color filter and the images are superimposed at the screen

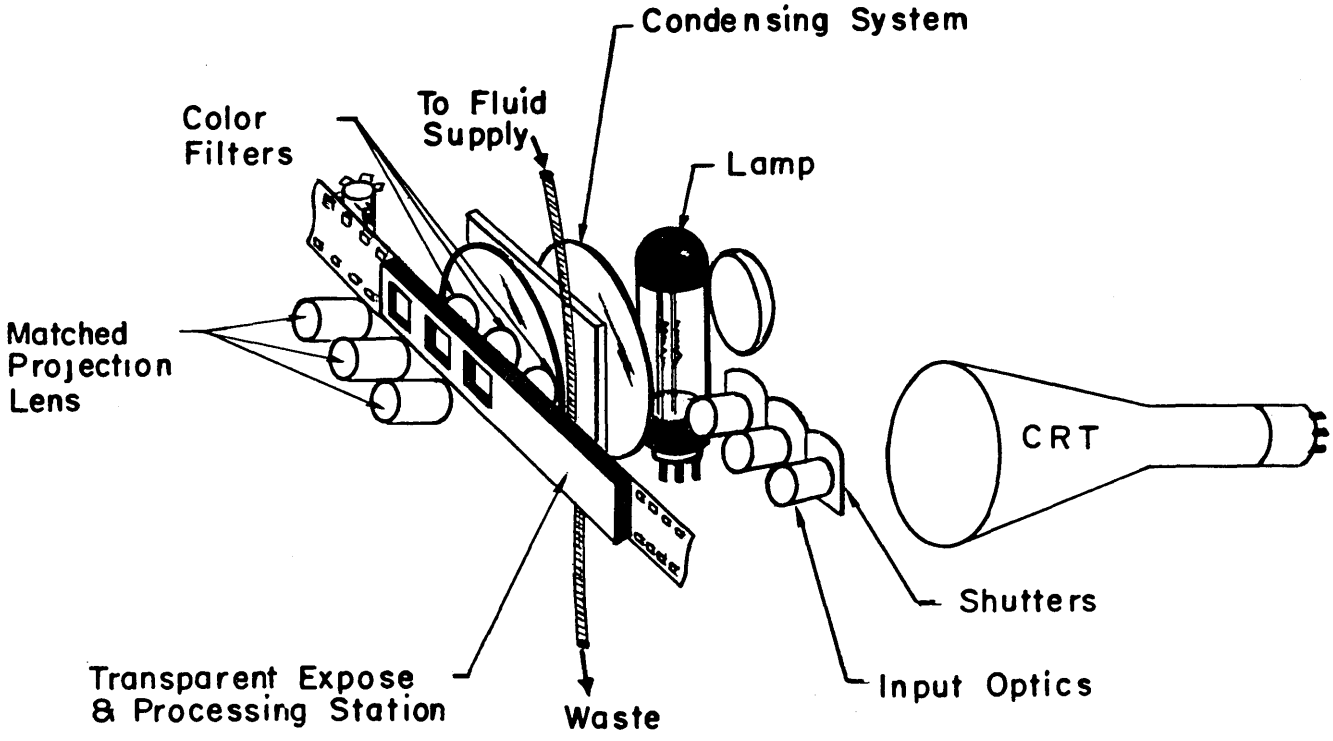


Figure 1—Multi-color film processor and projector

to form a single multicolor image. To properly control the color of data, each camera has a shutter which is opened or closed, depending upon the color instruction from the computer. For example, to generate the green portion of a multicolor image, the proper shutter is opened, "green" data is written on the CRT, exposing the "green" frame on the film. Then after the other exposures have been made, the image is processed and projected.

The other colors are formed in the same way. For multi-component colors, more than one shutter is opened for a particular exposure.

Silver halide emulsion is the most widely used light sensitive medium for recording the CRT image. The absolute sensitivity of silver halide is such that character exposure times of 50 to 1000 microseconds can be used. It can be seen from Figure 2 that the spectral response of a blue-sensitive film such as Ansco Hyscan is relatively flat out to 500 millimicrons, and well matches the output of the P-11 phosphor which peaks at approximately 460 millimicrons.

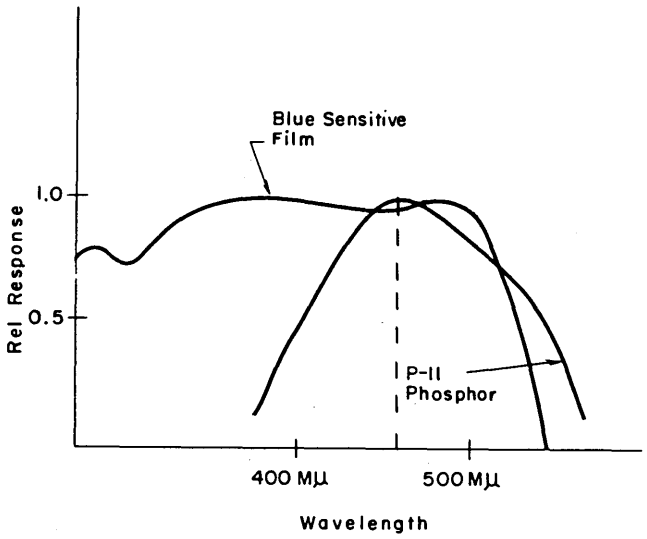


Figure 2—Film phosphor spectral characteristics

The demand for rapid response in computer-type displays has been met by the development of rapid processing techniques. The process consists of the use of special films containing hardened, large grain emulsions which make them suitable for processing at elevated temperature without appreciable loss in image quality. Total processing times of five to ten seconds can be achieved with developer temperatures of 130° to 140° F. The negative image produced in conventional film processing is not suitable for projection in a color additive process and therefore reversal processing is used to obtain a positive image. The steps involved in reversal processing are: (Figure 3)

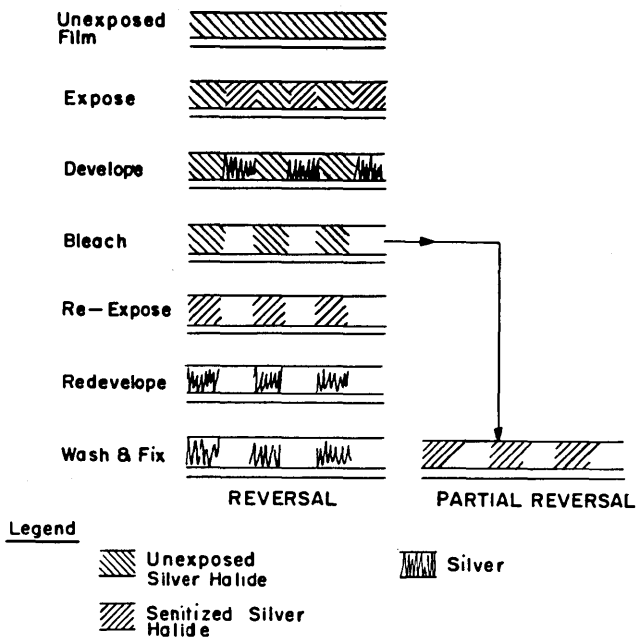


Figure 3—Reversal processing

Develop—Reduces the exposed halide crystals to metallic silver and leaves unexposed areas unreduced. The light energy absorbed by the exposed crystals increases their sensitivity to the developer and consequently they are reduced before the unexposed crystals.

Bleach—Converts the metallic silver into a water soluble compound so that it can be washed from the base material.

Wash—Removes the soluble products of the bleach process and leaves the exposed areas of the film clear, and the unexposed areas, which contain silver halide, opaque.

The previous three steps constitute a partial reversal process. The final image is the clear information areas against the unexposed silver halide background. To obtain a full reversal process, the following steps are added:

Re-expose—Sensitizes the remaining silver halide by exposure to light.

Redevelop—Converts to metallic silver the exposed silver halide crystals.

Wash—Removes remaining chemicals.

Under zero ambient illumination conditions, the partial reversal processes are capable of producing an image contrast of better than 50 to 1, and full reversal processes a contrast of better than 100 to 1. In applications where the surface area to be illuminated is greater than 150 square feet, the increased contrast afforded by the full reversal process can compensate for the lower display brightness resulting from the large screen area and enhance display legibility.

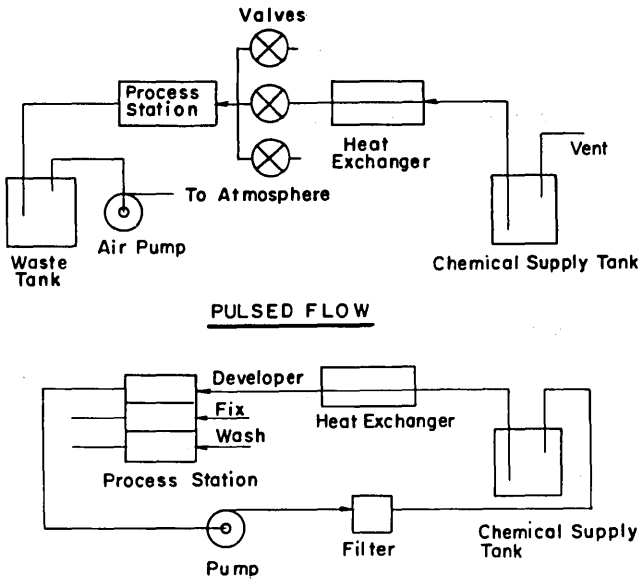


Figure 4—Basic fluid handling systems

There are two systems of chemical fluid handling currently in use; continuous flow and pulsed flow. Both of these techniques maintain the processing chambers below atmospheric pressure to lower the probability of leaks.

In the continuous flow fluid system, the processing fluids flow continuously through separate channels and the film is moved across each head sequentially in such a way as to come in contact with the required chemical at the proper time. The supply tanks are then maintained at some temperature less than that required for processing to extend the life of the chemicals and a heat exchanger provides the capability of heating the chemicals to the proper temperatures as needed. With the pulse flow system, the chemicals required for processing of the film pass sequentially through a single processing head. Normally-closed solenoid actuated pinch valves are used to sequentially open the appropriate fluid lines and allow chemical flow, and then to close the line after the required quantity of chemicals have flowed into the head. This technique is more reliable since an air pump is used, and because only one head seal is required.

Table II is a tabulated summary of the pertinent performance parameters of three systems which utilize this technique for their display capability. The values of the individual parameters do not necessarily represent a maximum, but rather represent a set of performance characteristics which have been achieved.

This summary includes the practical results of the technique in two large screen applications and in a small screen application.

TABLE II

PERFORMANCE CHARACTERISTICS

	Full Reversal	Negative Positive	Partial Reversal
Projection Scheme	Rear	Front	Rear
Screen Size (Ft)	12×16	8 × 8	1×1
Projection Distance (Ft)	22±¼	28±¼	3
Incident Illumination (Ft Candles)	26	30	100
Fall-Off (%) of Center to Edge Brightness	22	23	25
Symbol Brightness (Ft. Lamberts)	13	15	50
Contrast Ratio (Zero Amb.)	150:1	100:1	30:1
Symbol Height (Inches)	1.63	1.40	0.210
Colors	Seven	Seven	Four
Color Fringing (Inches)	0.13	0.12	N/A
Resolution (Line Pairs Per Screen Ht)	940	910	1000
Linearity (%) of Screen Width	0.5	0.5	0.5
Registration (%) of Screen Width	N/A	1.0	0.7
Response Time (Sec)	10	15	15

In summary, film systems can provide high quality, colored, large scale displays of almost any size. They are simple to integrate into data processing systems and present no unusual demands on the data processor. They are mechanical devices and utilize corrosive chemicals heated to high temperatures. Film systems by their very nature are static displays and best applied to tasks such as status monitoring and other functions where the 10 to 15 second update delay is not objectionable. They are available off the shelf from many manufacturers.

2. Scribe systems

Scribe projectors can provide the dynamics lacking in the film approach to large scale displays. These devices resemble a miniaturized x, y plotter. They utilize a servocontrolled stylus to scribe lines through opoque metallic coating on a transparent base material (Figure 5). The resultant image is then projected by a conventional optical system which resembles a 35mm slide projector. Some form of slide storage and access is usually provided. The scribe projector is usually of modular design; projection optics, light source, filters,

and scribing mechanism can be interchanged and configured to meet many requirements. Prepared slides can be substituted for the scribed slide and projected for use as background material. A cursor can be substituted for the scribing stylus and various combinations of light source and projection optics are available

An analog character generator and a manual input device may be included in the system. The manual input device will include a plotting surface and an auxiliary alphanumeric key board.

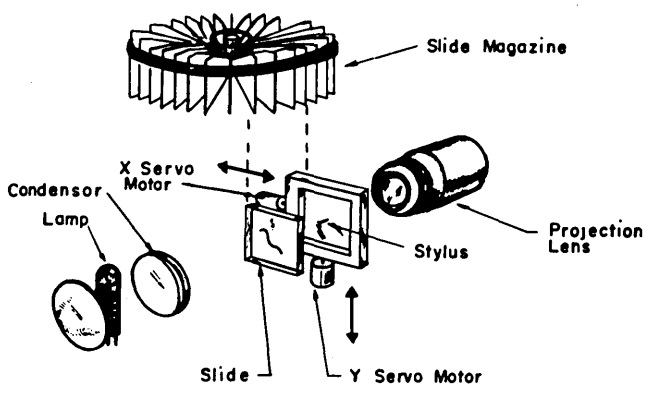


Figure 5—Plotting projector

In an operating system, a group of plotting projectors generally share a common set of control electronics. Combinations of plotting, spotting (cursor), and background projectors can be used. Two projectors are generally provided for each color required. This is to allow one active projector for current data and one idle projector to be used as follows. When the display on the active projector becomes too cluttered and must be changed, a limited history of past events is transferred to the idle projector. This projector is now activated and continues to plot, and the previously active machine is shut off. The slide can be then changed with no blank display time and the cycle repeated.

Table III is a brief resume of the typical performance that can be expected of currently available scribe system.

Scribe systems can provide dynamic displays of high quality. They are slow devices and can be interfaced with most remote communications channels. Because of their versatility, they can be configured to meet many applications. They are highly mechanical devices and contain many moving parts which limits the reliability that can be expected of the device. Scribe systems are best applied where the total amount of data is limited and where their dynamic characteristics are desired.

TABLE III

TYPICAL SCRIBE SYSTEM PERFORMANCE

Stylus Slew Time	60 miliseconds (Full Scale)
Symbol Scribing Time	
Random Position	10 Symbols per second
Adjacent Position	20 Symbols per second
Stylus Positioning	
Accuracy	.1% Screen Height
Repeatability	.03% Screen Height
Slide Storage Capacity	40 Slides
Scribing Area on Slide	1 in x 1 in
Resolution at Screen	1000 line pairs (1.5 mil stylus)
Slide Changer	
Time to Adjacent Slides	500 milliseconds

3. Projection CRT's

Projection CRT's have been utilized mainly in the TV mode of operation. They have been widely used in simulators and trainers to portray to the operator the world as it would appear to him in an actual situation. In this application a computer would control the presentation by controlling a camera which viewed a model of the terrain to be pictured. Projection TV systems have also been utilized to display computer generated data. Here scan conversion or direct digital conversion of data is required to provide a TV format.

Two types of optical systems can be used with projection CRT's. Schmidt optics (Figure 6) provide the best collection efficiency, but have poor resolution and high distortion. Refractive optics are of better quality and can be highly corrected for a given system. Their optical efficiency is lower and their cost higher than Schmidt systems. The CRT's used for projection are 5 or 7 inch tubes and are operated at voltages in excess of 30 KV. Voltages as high as 80 KV have been used for some applications. In this range, X-rays can be a serious factor. Life of the projection CRT is severely limited by degradation of the phosphor and darkening of the face plate due to electron bombardment.

Typical performance of a Schmidt TV projection system is:

Resolution	600 TV Lines
Light Output	200 Lumens
Screen Size	5 ft to 15 ft
Throw Distance	10 to 25 ft.
CRT Voltage	40 KV
Tube Life	500 hrs.

Projection CRT's have been used for direct dis-

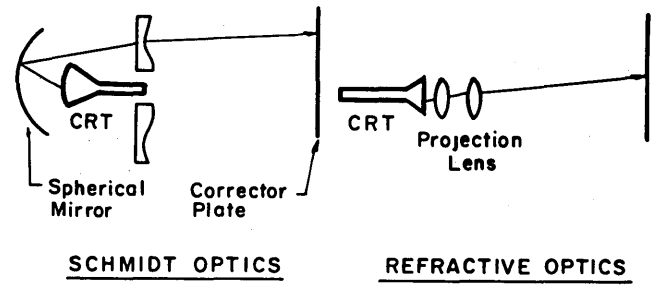


Figure 6—Projection CRT systems

play of computer generated data, but the results to date have been poor. The resolution of available CRT's is not high enough for this mode of operation. Also protection of the phosphor and face plate from damage by a static or slowly moving high energy electron beam has been difficult.

The low level of general performance has limited the use of projection CRT's for large scale displays. Their use has been confined mostly to the display of pictorial information.

4. Light valve systems

Another approach to the large screen projection of TV images is the "Light Valve" technique in which a control medium is used to control the transmission of light from a light source to the screen. The "Eidophor" is an example of such a device and uses a thin film of oil as the modulation medium. Figure 7 depicts the basic "Eidophor" operation. In the "Eidophor" a thin film of oil is continuously applied to a rotating spherical mirror and mechanically smoothed to a thickness of .1 millimeter.

The collimated light from a high pressure short are Xenon lamp source is reflected off the bar system and onto the oil surface which is contained with a vacuum tight chamber.

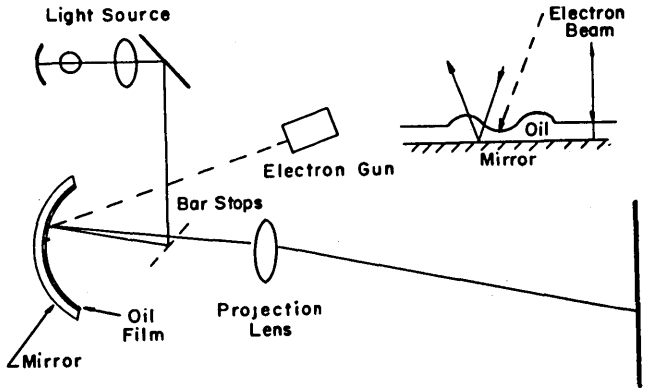


Figure 7—"Eidophor" light valve projector

An electron beam impinging upon the oil surface causes deformation to form in the control layer. Light passing through the deformed layer is refracted, reflected off the spherical mirror and refracted again; thus by-passing the stop and reaching the projection lens to be imaged on the screen. Light striking the non-deformed region of the oil film is imaged back upon the stops and does not reach the projection lens. By changing the size of the electron beam, the shape of the deformation can be changed and the amount of light passing through the system controlled.

Typical operating parameters are as follows:

Resolution	1000 TV Lines
Light Output—525 line B&W	4000 Lumens
Lamp Size	2500 Watt Zenon
Bandwidth	Up to 30 MC
Screen Size (ft)	9×12 at 50 ft projection distance
Throw Distance	50 ft.
Linearity	1% screen width
Cathode Life	100 hrs

To utilize the Eidophor—to display computer generated information, the data must be converted into a form compatible with the TV mode of operation. This can be accomplished in many ways. Some examples are:

a. View with a camera, the display being created on a small CRT operating on the conventional random writing mode.

b. Utilize a double ended electrical-in electrical-out storage tube. In this device, the display would be written onto one side of a storage target in the random mode and scanned off the other side of the target in the TV mode.

c. Utilize a digital data converter which, in real time, converts digital descriptions of the display into video signals compatible with TV operations.

While the Eidophor overcomes the limitation of brightness and resolution, it still requires data conversion to effectively display computer generated data. The present systems contain not only mechanical systems to distribute the oil but also have a complete, continuously operating vacuum system which includes a mechanical pump and an oil diffusion pump. These mechanical components coupled with the short life of the electron gun cathodes limit the applications to situations where large amounts of periodic maintenance can be performed and where continuous operation for long periods of time is not desired.

CONCLUSION

While large scale display techniques have advanced considerably in the past few years, these is still much room for improvement. Their capability to handle dynamic data needs considerable expansion. Cost which is now high must be lowered and reliability needs improvements.

Toward this end, considerable research is now underway to improve existing techniques. New films which do not need wet chemicals are being explored along with novel methods of processing conventional films. Considerable effort is under way to improve the performance of the light valve technology. Also, new and better techniques are being developed to convert digital data to the analog form necessary for the exploitation of TV type devices.

Some of the specific techniques under study now are:

a. The use of the "Bimat" process where the processing chemicals are contained in an absorbtive web material, processing is accomplished by placing the film to be developed in contact with the chemical saturated web and maintaining this contact for a period of time. A conventional negative and a positive result from this system.

b. Photochromic flim, a reusable UV sensitive recording media has progressed to the point where prototype equipment is being designed.

c. Laser displays have been under study for some time. Results at this time are not conclusive and it is doubtful if any application of the laser to large scale display is in the near future.

d. Electroluminescence has been under study for many years. While much progress has been made, we are still far from applying this technology in practical systems, but electroluminescence has reached the point where it can be considered for use as discrete indicators.

With the concerted research directed at improving the display art and the increased demand for large scale displays generated by the growth of data processing into the higher echelons of management, we will see a dramatic expansion of the use of large scale displays.

The CRT display subsystem of the IBM 1500 instructional system

by R. H. TERLET
International Business Machines Corporation
Los Gatos, California

INTRODUCTION

The IBM 1500 Instructional System is an experimental system for computer-assisted instruction, designed to administer individual programmed lessons to 32 students at once. Working through one or more teaching devices at his own instructional station, a student may follow a course quite different from, and independent of, lessons presented at other stations. Instructional programs stored in central files control lesson content, sequence, timing, and audio-visual medium, varying all of these according to the student's responses.

Briefly, the system works like this: The processor retrieves instructional material from the files and presents it on a station input/output device. The student responds as directed. The processor then compares his response with the answers anticipated in the instructional program and continues with the next lesson material or branches to remedial instruction. The system can keep records of student answers, response times, and accuracy.

The IBM 1500 Instructional System is shown in Fig. 1a. The central processor, an IBM 1131, has access to core storage of 32,768 sixteen-bit words and a cycle time of 3.6 or 2.2 microseconds. Under direction of the 1500 Operating System, the processor controls the time sharing of the student stations (Fig. 1b) and the execution of the instructional programs. The IBM 1502 Station Control multiplexes the input and output of the station devices, which may include an input/output typewriter, an image projector, and an audio play/record unit, as well as the CRT display to be discussed. Each CRT display unit includes a keyboard for student responses, and may include a light pen as well. The 32 CRT display units, the station control and the processor core storage together make up the display subsystem.

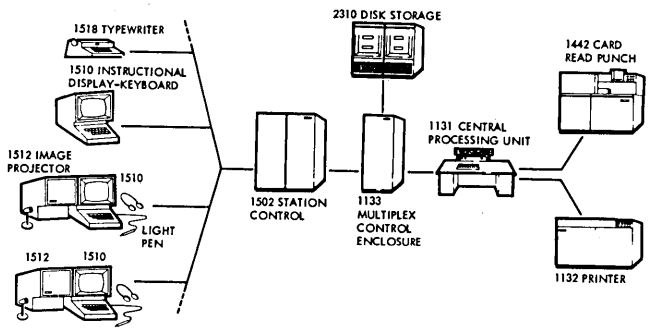


Figure 1a—The IBM 1500 instructional system

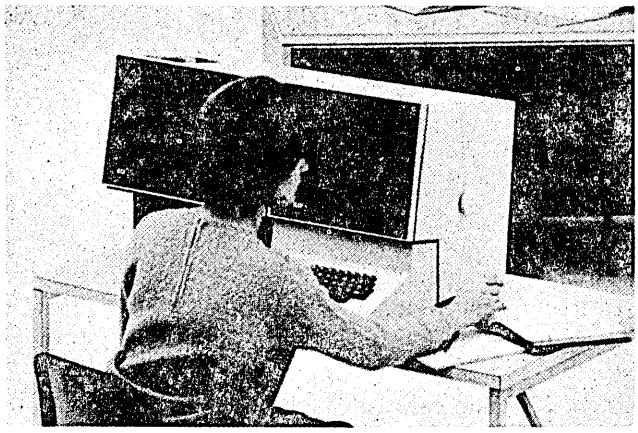


Figure 1b—The IBM 1510 Student station

CRT speed and flexibility make this type of display unit attractive as a basic instructional device. In some ways, however, the display requirements for instruction are more demanding than those for conventional data processing applications.

A character set for instruction must often include a far larger number of characters and symbols than is needed for other applications. For example, teaching

a foreign language may require that two different alphabets be displayed at once. In mathematics, a display must often include exponents, subscripts, and fraction lines, as well as alphanumeric characters and mathematical symbols.

An instructional display must also be unusually flexible. A student should be able to complete displayed sentences and to insert words within a sentence. It should be possible to display combinations of simple images and printed text in teaching certain concepts. To meet the needs of different student stations and achieve a stimulating variety in lesson presentation, character fonts should be easily changed.

Finally, to give course authors freedom in varying the mode of lesson presentation, the CRT displays should be in a form compatible with the alternative typewriter printout.

These system objectives were achieved in a versatile display system capable of handling 32 student displays. The CRT display units are of conventional design,[1] with a magnetic disk buffer to store and refresh the images. Characters and images are positioned under program control. The system is able to handle any number of large character sets by means of program-changeable fonts ("dictionaries") placed directly in core storage, where they are accessible to the character generation logic and to the system program. Allocation of space for the fonts reduced the available core storage, already a good deal smaller than is common in multiterminal time-sharing systems. The problem of satisfactorily sharing the relatively small core area remaining was solved by the application of data chaining techniques in core storage. Here the high interrupt servicing overhead usually needed to chain blocks of data was avoided by making such chaining a hardware operation. These solutions depended on a flexible manipulation of core storage made possible by the storage access channel feature of the processor.

Hardware description

The major components of the Display Subsystem, shown in Figure 2, are (1) the CRT display, keyboard, and light pen at the student station; (2) the display control and light pen adapter of the station control; and (3) the core storage of the processor.

Lesson material called from the disk storage is interpreted by the Operating System and, under the direction of the display control logic, is translated from a stream of computer-coded characters and symbols into a sequence of the appropriate displayable dot patterns. These patterns are obtained from the dictionaries which occupy a portion of the core storage. As they are translated, the dot patterns are

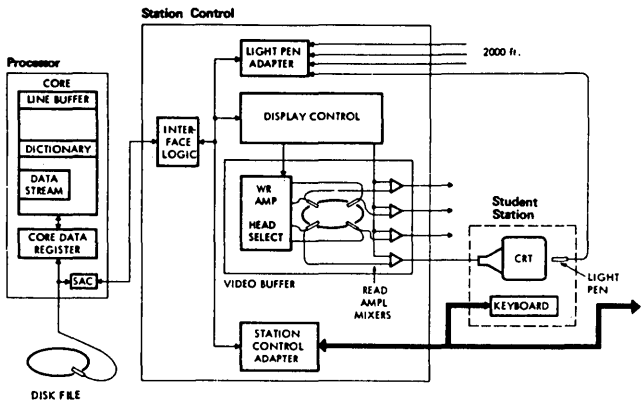


Figure 2—The IBM 1500 display subsystem

temporarily stored, a line at a time, in the line buffer, before transfer to the video buffer. The video buffer is a magnetic disk assembly in which a separate recording track, with an associated fixed read/write head, is assigned to each display unit. This track can store one complete frame of text and image material, already coded in the video dot pattern, which can be read again and again to maintain a continuous display.

The display control logic also generates timing signals that control the vertical and horizontal synchronization of the CRT displays and, during student light pen responses, serve to identify the pen position for the light-pen adapter.

The display unit consists of the CRT display, the keyboard, and the light pen. The CRT display presents instructional text and images to the students. The keyboard is the major input device for student entries, which are in most cases immediately displayed on the CRT under program control. The light pen is an optional input device with which the student can point to selected response areas on the face of the CRT.

The general characteristics of the CRT display are summarized in the table. The display area on the screen is 4.8 inches high and 8 inches wide. The control logic divides this area into 40 columns and 32 horizontal half-lines.

Each character is based on an 8 by 12 dot matrix that is one column wide and two half-lines high. The system logic does not allow for space between columns or half-lines; thus dot patterns can be joined to form continuous lines, while needed space can be written into the display code.

The display electronics are of conventional television design, with some special attention to linearity of the vertical and horizontal sweep circuits.

The video buffer consists of a specially assembled pack of six IBM 2316 disks, with 32 magnetic heads,

Horizontal Visible Dots	320
Vertical Visible Lines	192
Horizontal Frequency	6.5 kHz
Video Digital Frequency	2.5×10^6 pulses per second
Vertical Frame Rate	30 Hz
Interlace Ratio	2/1
Visible Dots	61,440
Character Matrix	8 x 12 dots
Display Size	8 x 4.8 inches
Geometric Distortion (Initial Settings)	Vertical Line ± 0.06 inch Horizontal Line ± 0.01 inch
Linearity (Deviation from Nominal)	Center Third: ± 15% Outside: ± 25%

General characteristics of the IBM 1500 instructional display unit

in four assemblies of eight each, mounted around the disks as shown in Figure 3. The heads are fixed in position, and each one reads and writes a specific

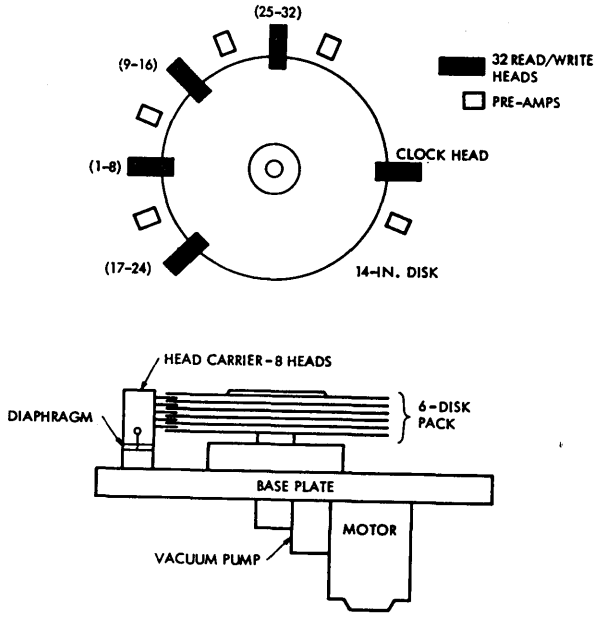


Figure 3a – The video buffer, top and side view

track assigned to a given student display. Thirty-two tracks on the outer edges of the 14-inch disks are chosen to give longer and approximately equal track lengths.

The video buffer stores and regenerates the images displayed on the CRT. The information on the video buffer tracks is recorded in the non-return-to-zero (NRZ) mode. The digital recording is a one-to-one image of the dot patterns being displayed on each instructional unit.

The disks rotate at 1800 rpm, and one revolution of the disk corresponds to one frame on the display. The CRT scanning is synchronized with the rotation

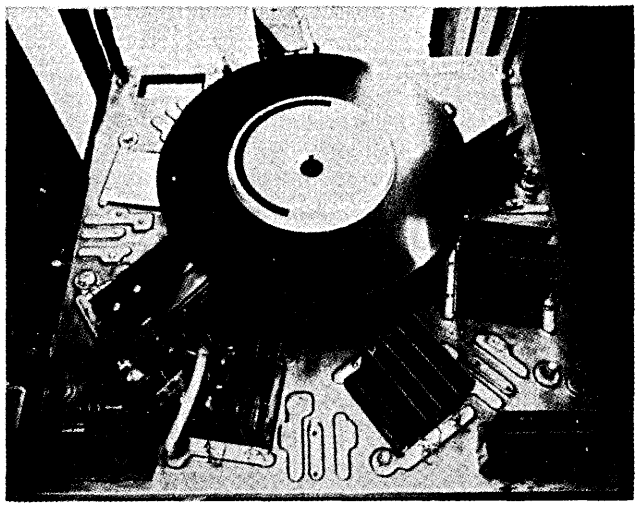


Figure 3b – The video buffer (photograph)

of the video buffer disk pack, so that the format of the data on the track corresponds to the scanning of the face of the CRT, as shown in Figure 4. Since each frame of the display consists of two interlaced fields, half of the data track contains the odd scan field data, while the other half contains the even scan field data.

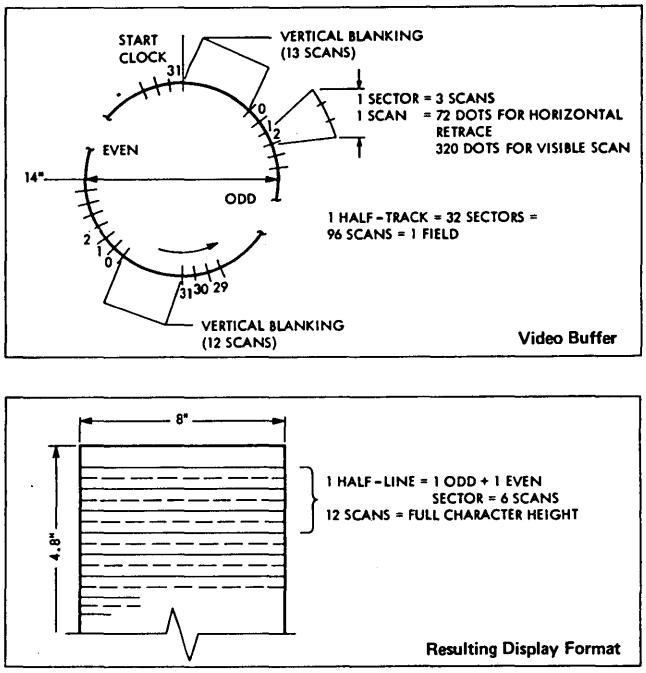


Figure 4 – Data format on video buffer and resulting display format

The 192 visible scan lines in a frame are addressable in 32 half-line segments of six scans each (automatically divided into three even and three odd scans, which are stored on different halves of the data track). A complete line of characters then represents 12 scans on the CRT, six in the odd field, and six in the

even field. Each half-track, or field, consists of 96 visible scans, plus 12 scans for the vertical retrace in the odd field, and 13 scans for the vertical retrace in the even field—the extra scan here allows the field interlace.

Each scan line in the video buffer consists of 392 dots (or bits). The first 72 are always blank to allow for horizontal flyback time; the remaining 320 dots correspond to the 40 columns of displayable data in a horizontal sweep of the CRT.

In addition to the data tracks for each display unit, the video buffer contains a separate timing track read by its own fixed clock head. The timing track stores a steady sequence of clock pulses with a blank home gap that serves as a reference for display sync signals and disk read/write functions. The clock pulses increment digital counters, whose decoded output supplies signals that control display blanking and vertical and horizontal sync. Since the display data in the video buffer is written on the disk under control of the same data clock, the buffer output and CRT display are thus properly synchronized.

The video buffer contains the necessary electronics for head selection, writing, and sync mixing. Each head is in a read (or display) mode except when writing, and a single write driver serves all the heads.

The display control logic circuits are time shared for video buffer recording and character generation.

A single command to the video buffer control logic can change any track (1) by erasing (i.e., rewriting with blanks) the complete track, or (2) by rewriting or erasing:

(a) a full line (12 scans),
(b) a half-line (six scans),
(c) a word or phrase (full or half-height) within a line.

Insertion of single words within a line is simplified by our choice of the NRZ mode of digital recording, with a binary magnetic state directly corresponding to black and white dots on the display. The write circuits and control logic were designed to insert short dot patterns on successive segments of a track without disturbing previously recorded data.

The display unit uses a standard IBM keyboard. The station control is an I/O device multiplexer operated under program control of the processor. At regular intervals the adapter polls the I/O devices and reads in any keyboard input. Under I/O servicing program control, the student's input message is assembled and the character he keyed is displayed on his CRT. In this way the computer program positions the displayed student response and selects the character font in which it is displayed. This means that a keyboard character set can be changed by the program. The student keyboard itself could be changed by an overlay.

The subsystem response time to keyboard inputs is a function of the length of the queue to the video buffer. The length of the queue in turn is related to the input keying rate of all student stations. The response time then can vary from 20 msec to a worst case of 1.07 sec, 200 msec being typical.

Character generation

The Display Control logic basically adds two special instructions, "translate" and "transfer," to the repertoire of the processor. A sequence of translate and transfer instructions records a new display frame on the video buffer, ready for immediate continuous display.

We can trace this process by looking again at Figure 2. The character generation logic operates on a string of characters (the data stream) in core storage. It translates the computer-coded characters into video-coded dot matrices by a table lookup technique—finding the proper dot image in the designated dictionary, always at hand in core storage. The character generation hardware then stores the translated video code in a reserved area in core: the line buffer. The line buffer holds one full line (12 scans) of video information.

When the CPU program gives a transfer command, the line buffer contents are recorded on the selected track and sector of the video buffer. The execution of the transfer command automatically clears the line buffer on readout.

The data stream may be of any length, but is always organized in blocks of 32 sixteen-bit words. These blocks are chained by familiar list processing techniques; that is, the last word in the block is the address of the next block. In this case, though, programming trouble and processing time were saved by "wiring in" the repetitive subroutine necessary to link the blocks. No programming intervention is needed to determine the new address; the logic automatically reads in the address and continues the processing of the data stream from the new core location.

Each dictionary occupies 768 consecutive core locations. The number of dictionaries in the system is a user option. Location of the dictionaries directly in core, readily accessible to the character generation logic, was a choice dictated by the need for flexibility in changing fonts in an instruction system. Different dictionaries can be used in one line of text. The dictionaries can be loaded, altered, or switched directly by program.

We could afford to use a fairly large proportion of the relatively small available core storage for this

important function because list processing of the data saved space. And we could afford to use list processing in the data stream of a multidevice time-sharing system because automatic data chaining by the hardware saved time.

The line buffer is a fixed area of core, capable of storing 240 words, It too can be directly loaded by program.

The character shapes are specified by the user when he constructs a dictionary. Each character or image pattern takes six words of storage, enough to describe an 8 by 12 dot matrix. Characters are usually constructed in 7 by 10 matrices, leaving one blank space between characters and two blank scans between lines on the display. The 7 by 10 matrices can adequately describe many kinds of characters sets, including upper and lower case alphabets. For graphics or very large characters, basic patterns are usually specified in 8 by 12 matrices which can be combined to form larger patterns.

Characters are generated in a "cycle steal" operation. The display control logic "steals" a cycle from the core storage through the storage access channel. This feature gives the display control random access to all core areas needed for character generation. It was decided to let this operation take all the processor memory cycles it needs until the whole data stream is completely translated. This had the effect of reducing appreciably the programming interrupt overhead associated with the displays. Storage utilization is the same as if the processing was overlapped, but no interrupt servicing is required because none of the processor registers are changed at the end of the execution. File operations were given a higher "cycle steal" priority so that system performance did not suffer.

Each code of the data stream is scanned in succession. A character code is translated and the appropriate bit pattern stored in the line buffer; a function code will usually modify a control counter. Translation relies on one dictionary until a dictionary change code appears in the data stream.

Translation stops at this point, and the program can intervene to specify the next dictionary address. This program intervention allows flexible assignment of dictionary areas in core storage and relieves the course author of the problem of addressing dictionaries in a multi-user environment.

A character is translated as shown in Figure 5. After initialization of the display control hardware, one data stream code is read in (1). The code is combined with the dictionary address to access the dictionary (2,3). The character generation hardware reads in the 16-bit dictionary word it finds (4), writes the eight high-order bits in the line buffer (5), and increments the line buffer address counter one scan position (6). Next it stores the eight low-order bits in the line buffer, and again advances the line buffer address counter.

For each character, the dictionary is accessed six times and 12 partial scan lines are stored in the line buffer. At the end of this translation the line buffer address counter points to the next character column.

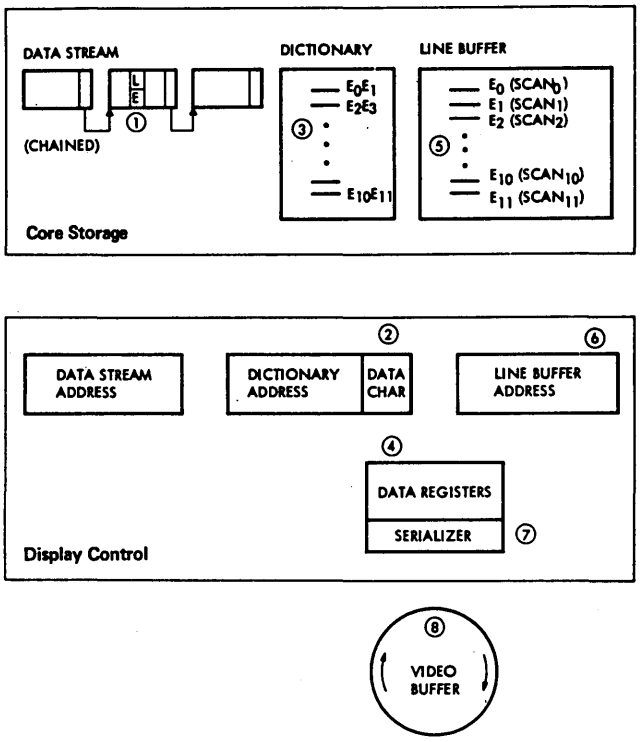


Figure 5 – Character translation

A transfer command causes the information in the line buffer (5) to be read by the display control logic (4), serialized (7), and written in the designated track and sector of the video buffer (8).

A transfer command may result in transfer of a full line (Figure 6a), a half line (Figure 6b), or a few columns within a line (Figure 6c), as in word insertion.

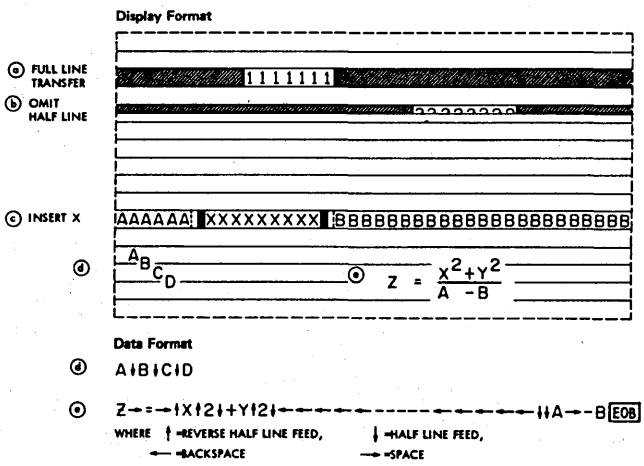


Figure 6 – Variable spacing with function codes

The video output to the selected display is blanked only for the line that is being recorded. The blanking extends over a line and a half to blank out the transients induced by the write circuits in the read amplifier. The blanking lasts 2.5 milliseconds and is not distracting to the person viewing the display.

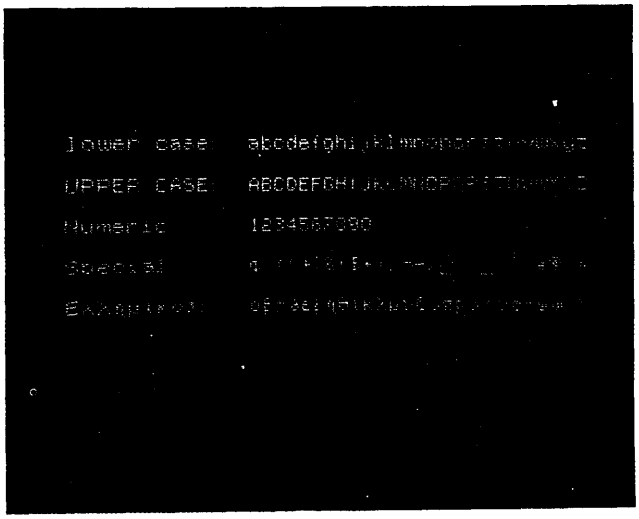
Since storage in the line buffer is basically an OR operation, characters can be superimposed on others (as in underlining or placing accent marks over letters) by combining the new character with a previously stored character.

The character generation logic also can handle subscripts and superscripts, as well as lines of text, formulas, and graphic images extending over more than one line of the display. Four function codes give the flexibility needed for this kind of variable spacing (see Figure 6d and 6e): half-line feed and reverse half-line feed to move a character down or up half a line (e.g., for subscripts and superscripts); space and backspace to move a character one column to the left or right. With these function codes, the data stream in the instructional system can have the same basic format for CRT displays and typewriters—the two basic output modes; and input formatting is simplified.

The vertical displacement codes alter the scan reference count in the line buffer address counter, while the horizontal displacement codes alter the column count. The original address stored in this counter is specified by each translate command.

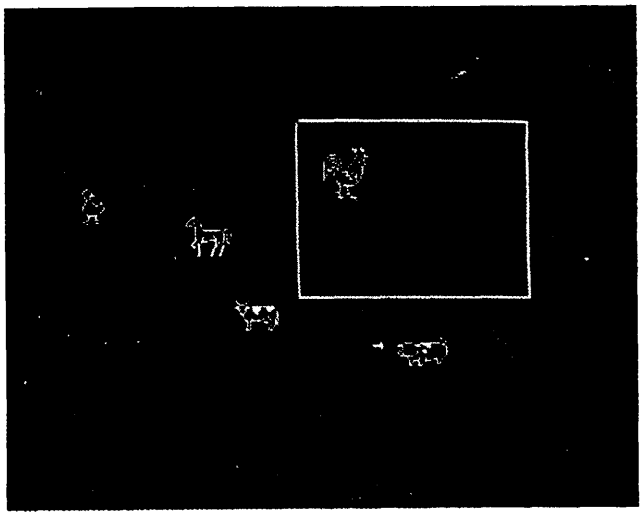
Since the line buffer stores just one line (12 scans) at a time, it is necessary, in writing an expression that extends over many scan lines, to give one translate and one transfer command for each line.

Figure 7 shows how characters and graphic images constructed in this way actually appear on the face of the CRT.



The light pen

An optional feature on the CRT display is the light pen, a pen-shaped device that lets the student respond directly to the displayed instructional material.



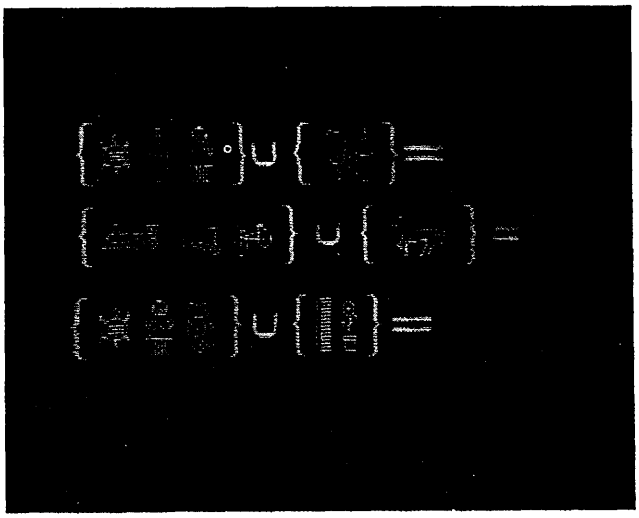


Figure 7—Sample CRT displays

The pen is a light-sensing device that transmits signals to the light pen adapter.

To select a response from a number of alternatives displayed on his screen, the student presses the tip of the pen against a designated target area. The required depression of the pen tells the system that the student is actively responding (and not just moving the pen over the face of the display) and gates the detection circuits that will transfer position coordinates of the selected target to the program.

Light pen operation may be understood from the diagram of Figure 8. Depressing the tip closes a switch within the light pen. The photodiode can now detect

the phosphor light output as the electron beam sweeps across the selected target on the face of the CRT, and the station is in an enter mode. In this mode, horizontal and vertical sync signals from the CRT deflection circuits are gated back to the light

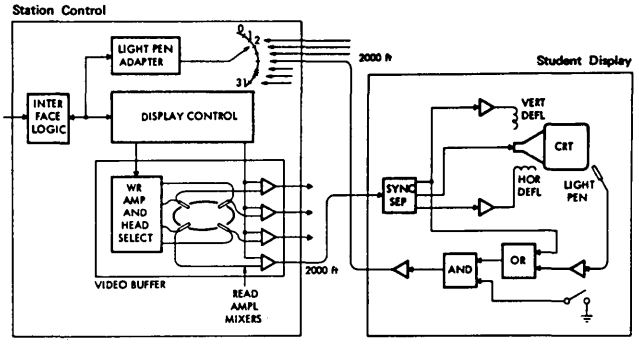


Figure 8 – CRT light pen operation

pen control logic, and the amplified photodiode pulses are gated to the same coax line.

The horizontal and vertical sync signals from the display control unit are used by the control logic to determine that the light pen is in the enter mode. The horizontal sync pulses also reset and start a counter that counts timing pulses (from the video buffer clock track) until a photodiode detect pulse is received. The number of timing pulses between the horizontal sync pulse and the photodiode pulse (see Figure 9) gives the horizontal position of the pen on the face of the CRT. The vertical position is obtained by copying the display control half-line counter at the time the photodiode pulse is received.

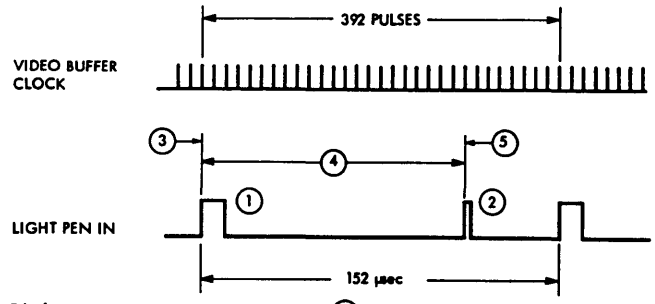


Figure 9 – CRT light pen timing

One control unit handles input from the light pens at all 32 student stations through time multiplexing. The control logic scans all the pen input lines in succession until it detects the sync signals that identify a station in the enter mode. The logic then waits for the detect pulse, determines the light pen position, and interrupts the processor to transfer the position coordinates to the program. Once the interrupt is serviced, the logic automatically resumes the scanning operation. Provision is made to avoid tying up the control circuits if a light pen should accidentally be depressed against a nontarget area of the CRT or some other surface. If no detect pulse is received during a full revolution of the disk buffer, scanning automatically resumes. The resolution of the light pen is an area one column wide and half a line high.

The display subsystem is capable of servicing one light pen every 33 milliseconds, on the average. Since one light pen is serviced at a time, the servicing rate varies with the number of active light pens.

CONCLUSION

The objectives were to design a very versatile display subsystem at a minimum system cost with available technology. To achieve these objectives, hardware and software design were closely coordinated.

Video buffering on magnetic disks proved to be an economical method of storing and refreshing 32 independent displays.

The design capitalized on the availability of a processor needed for the general system operation. User-definable, program-changeable fonts located in core storage met the need for many large character sets, capable of writing simple graphic images as well as alphanumeric text. The hardware design was both general (independent of the font) and versatile, giving a high degree of software control over the construction of expressions (e.g., accents, superscripts) and their positioning on the display (e.g., word insertion).

The following features were implemented in hardware to minimize subroutines size and associated processing overhead.

1. The number of interrupts was reduced by making the character generation a special instruction instead of a pure channel operation.
2. The limited availability of core space was eased by a hardware chaining of small core areas.
3. Finally, the hardware was designed to handle character streams in a manner compatible with conventional typewriter character streams. This had the double advantage of CRT-typewriter character stream compatibility and simplified formatting of the student inputs for the CRT display.

ACKNOWLEDGMENT

The CRT display system described, as an integral part of the IBM 1500 Instructional System, is a result of a cooperative effort of the engineers and programmers in the IBM Instructional Systems Development Department, to which many others elsewhere in IBM have also contributed. Special thanks must

be given to Kenneth Senne and P. A. Smith, who worked with the author on the logical design and implementation of the CRT display subsystem, and to M.G. Hurley, W. D Musson, and Dr. G. H. Royer, who were very helpful in the preparation of this paper.

REFERENCE

1 R AZIZ
An instructional display terminal
Proceedings of the Eighth National Symposium of the Society for Information Display May 24 1967 San Francisco California p. 83

Conic display generator using multiplying digital-analog decoders

by HOWARD BLATT
Massachusetts Institute of Technology
Lincoln Laboratory,* Lexington, Massachusetts

INTRODUCTION

The need has been recognized for a computer-driven generator which is capable of drawing complex curves *without* imposing a great burden on the central computer. It has been shown that analog[1] and hybrid techniques[2,3] offer promise of lifting some of this computational and storage load. Past approaches using these techniques have been limited in speed and in kinds of curves drawn. Roberts has proposed a hybrid analog-digital generator based on homogeneous coordinate mathematics[4,5]. Such a generator has been designed and built using the wideband multiplying decoder as its basic component.[6]

General two-dimensional conic sections are drawn by this generator by the artifice of generating parabolic curves in 3-space and dividing by one of the coordinates to effect a perspective transformation onto a plane parallel to the plane of the remaining coordinates which becomes the plane of the display. Circles, ellipses, and hyperbolas are perspective transformations of the parabola. The 3-space vector generated parametrically as a function of time is:

$$\bar{p} = [x(t), y(t), w(t)] \tag{1}$$

where:

$$\begin{aligned} x(t) &= x_0 + x_1 t + x_2 t^2 \\ y(t) &= y_0 + y_1 t + y_2 t^2 \\ w(t) &= w_0 + w_1 t + w_2 t^2 \end{aligned} \tag{2}$$

Division by w(t) yields the vector

$$v = \left[\frac{x(t)}{w(t)}, \frac{y(t)}{w(t)}, 1\right] \tag{3}$$

The direction of the vector remains unchanged by the division since all components are divided by the same quantity. The tip of the vector, however, lies on the plane $w = 1$. Fig. 1 illustrates this for a semicircle. The system described below is a hybrid analog-digital device which accomplishes the operations indicated in Eqs. 2 and 3.

*Operated with support from the U. S. Air Force

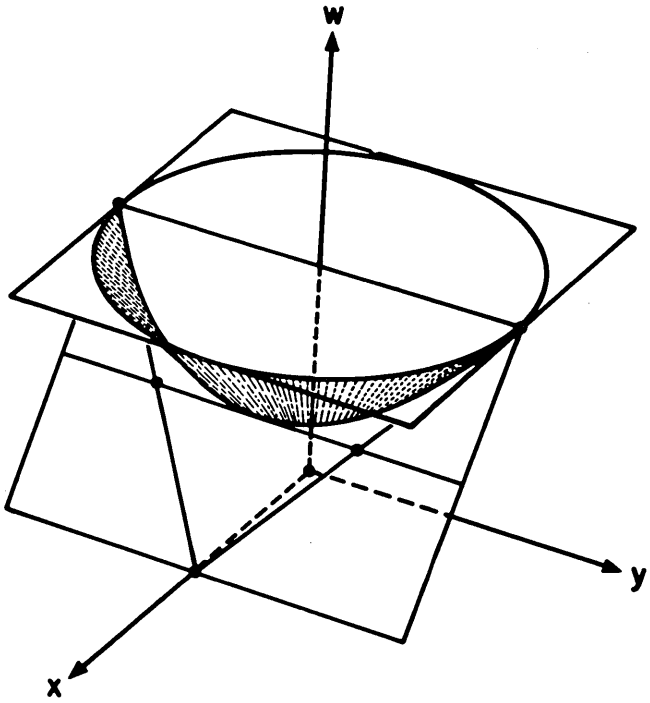


Figure 1—Projection of parabola into a circle in a homogeneous coordinate system

Operation of The Conic Generator

A block diagram of the system is shown in Figure 2. The blocks labeled t, w_0, w_2, x_1, x_2, etc, are multiplying decoders each of which produces an output equal to the product of analog voltage and a 10-bit digital number.* The multplying decoders are described in detail below (Figure 6 is the decoder schematic. The digital input for each decoder is held in a 10-bit buffer. The register holding the digital input for the two t-decoders is a 10-bit binary counter. To draw a curve segment the buffer registers holding the constants for the de-

coders (except the t-decoders) are filled by data transfers from the central computer. These are held constant for the duration of the segment. The t counter counts from 0 to 2^n ($O \leqq n < 10$) during a segment with the final count under program control. The t decoders are scaled so that the analog input voltages r and rt are multiplied by t ($0 \leqq t < 1$) over the counting range from 0 to 2^{10}. The frequency of the counter clock together with the total count determines the time to draw a segment. At the completion of a segment the counter is cleared. The buffers holding the constants, however, are not. For the next segment only those constants which have to be changed are affected.

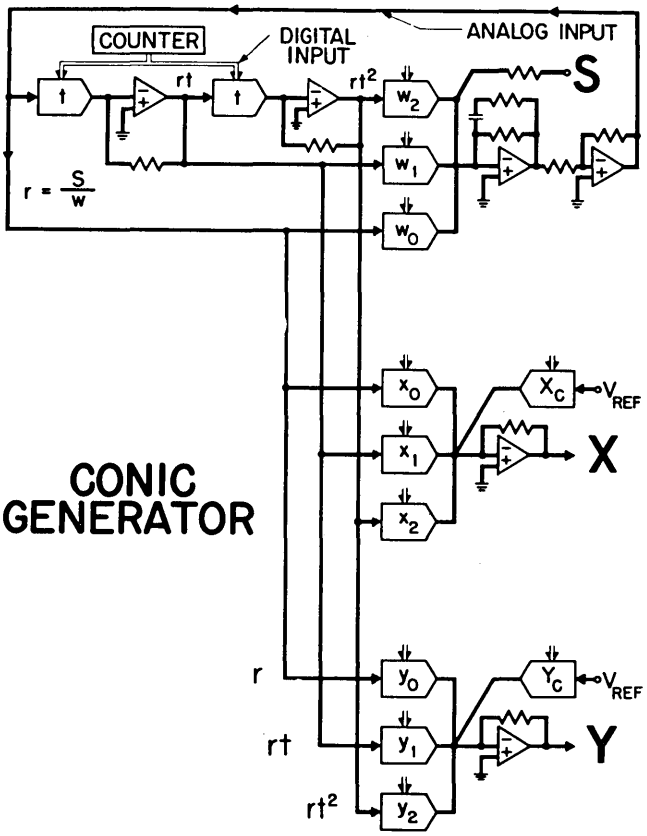


Figure 2—Conic generator block schematic

Consider the closed loop portion of the system. The effect of the high gain amplifier in the closed loop is to force the product of the analog voltage r and the digital number $w_0 + w_1t + w_2t^2$ to equal S. Or:

$$r = \frac{S}{w_0 + w_1t + w_2t^2} \tag{4}$$

Division by a second degree polynominal in the parameter t is thus accomplished. The analog signals r, rt, rt^2 so developed are multiplied by the digital numbers x_0, x_1, x_2, y_0, y_1, y_2 and translation constants Xc and Yc added to form the deflection signals.**

$$X = \frac{x(t)}{w(t)} + Xc = \frac{x_0 + x_1t + x_2t^2}{w_0 + w_1t + w_2t^2} + Xc$$

$$Y = \frac{y(t)}{w(t)} + Yc = \frac{y_0 + y_1t + y_2t^2}{w_0 + w_1t + w_2t^2} + Yc \tag{5}$$

A programming algorithm has been developed to determine the above coefficients. The range of w(t) is limited at the low end by the maximum output voltage of the amplifiers to +0.19. The range of w(t) during the drawing of a transformed conic determines how much of the conic may be drawn in one segment. For the limiting case of no variation in w where $w(t) = w_0$ (no time varying feedback) only straight lines and parabolas could be drawn. The other conics (as well as any other curves) would have to be pieced together with parabolic segments.

Two examples of conics generated by this display are shown in Figures 3 and 4. The circle was drawn as two semicircles:

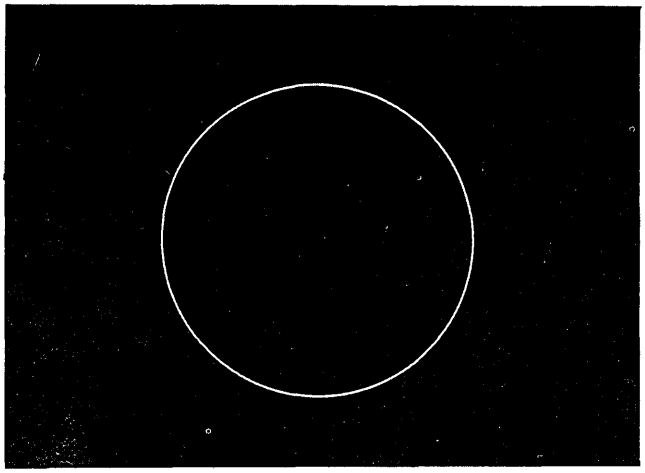


Figure 3—Circle drawn as two semicircles

$$X = \frac{-t+t^2}{0.5-t+t^2}, \quad Y = \frac{0.5-t}{0.5-t+t^2}$$

and (6)

$$X = \frac{t-t^2}{0.5-t+t^2}, \quad Y = \frac{0.5-t}{0.5-t+t^2}$$

The hyperbola was drawn in two segments:

*Xc and Yc are 11-bit decoders.

**The voltage S scales the entire picture and has been dropped from these and succeeding equations.

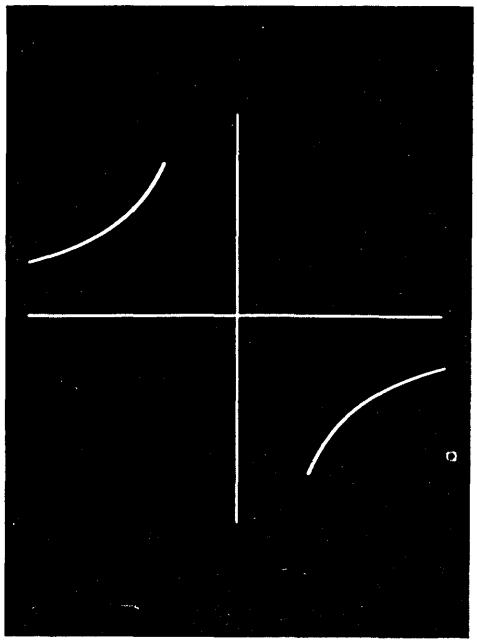


Figure 4—Hyperbolic segments and coordinate axes

$$X = \frac{0.25+t+t^2}{0.5+t}, \quad Y = \frac{1}{0.5+t}$$

and (7)

$$X = \frac{-0.25-t-t^2}{0.5+t}, \quad Y = \frac{-1}{0.5+t}$$

In both figures the range of t was 0 to 1 so that for the semicircles, $0.25 \leq w(t) \leq 0.5$, and for the hyperbola, $0.5 \leqq w(t) \leqq 1.5$. The clock rate was 1 MHz resulting in a 1 ms drawing time for each segment. The scale setting voltage S was set at 4 volts, which corresponds to a radius of 0.5 screen diameters. In order to draw more of a conic in one segment, the minimum value of w(t) must be allowed to get smaller. Since $r = \frac{S}{w(t)}$ the scale setting voltage must be made correspondingly lower in order not to saturate the amplifiers producing r. The signal-to-noise ratio varies directly, however, with the scale setting voltage S resulting in a tradeoff between length of conic segment and S/N. This is illustrated in Figure 5 where an elliptical segment is drawn with $w(t) = 0.28 - t + t^2$. (The x's and y's are the same for the semicircle above.) The minimum value of w(t) is 0.03 occurring at $t = 0.5$, and a maximum is 0.28 occurring at $t = 0$ and $t = 1$. The scale setting for this segment was reduced to $\frac{1}{2}$ volt to prevent saturation. The increased noisiness is apparent as is the variation in beam velocity.

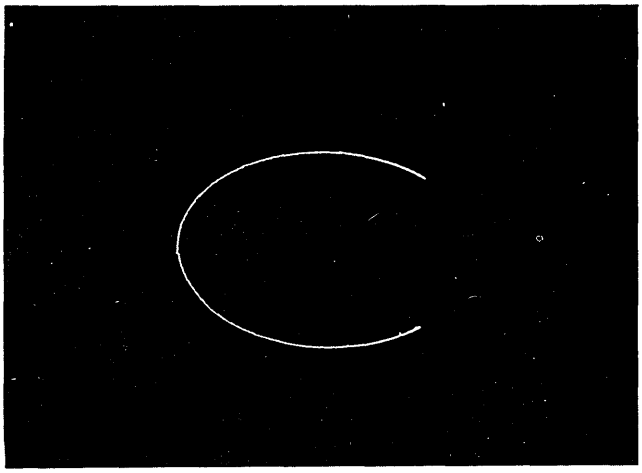


Figure 5—Elliptic segment

The data word format includes two 11-bit subwords for the decoder constants, four N bits which determine the t counter's maximum count, a preset bit described below, and 3 control bits for steering the 10- or 11-bit subwords to the proper buffer registers. These "destination" bits also determine which presets are to be made. Another bit is provided which tells the generator whether it should start or wait for more data. The N bits have meaning only for the data transfer immediately preceding the start of a segment.

In order to draw a curve segment the generator requires in the most general case 11 subwords (nine 10-bit and two 11-bit). However, in many cases of real interest only a few or even one subword has to be changed in going from one segment to another. An example is the generation of similar parallel segments. In this case a translation only is involved and one data transfer (Xc, Yc) is required. To facilitate the drawing of certain common curves a number of presets are provided in the control. Their use obviates a number of data transfers. For example, a preset for drawing lines sets x_2 y_2, w_1, and w_2 to zero and $w_0 = 1$. This preset occurs when the preset bit is a 1 and the destination code for (x_1, y_1) is given.

The multiplying decoder

The basic component of the conic generator is the multiplying decoder shown in Figure 6. Its operation is as follows: A positive analog signal, applied to the driving operational amplifier is inverted, offset, and applied to the resistor-diode network in the input path of the output operational amplifier. Digital levels, applied to diodes D'_0-D'_9 either divert the resistor currents or allow them to flow into the summing node which is kept at virtual ground by the output opera-

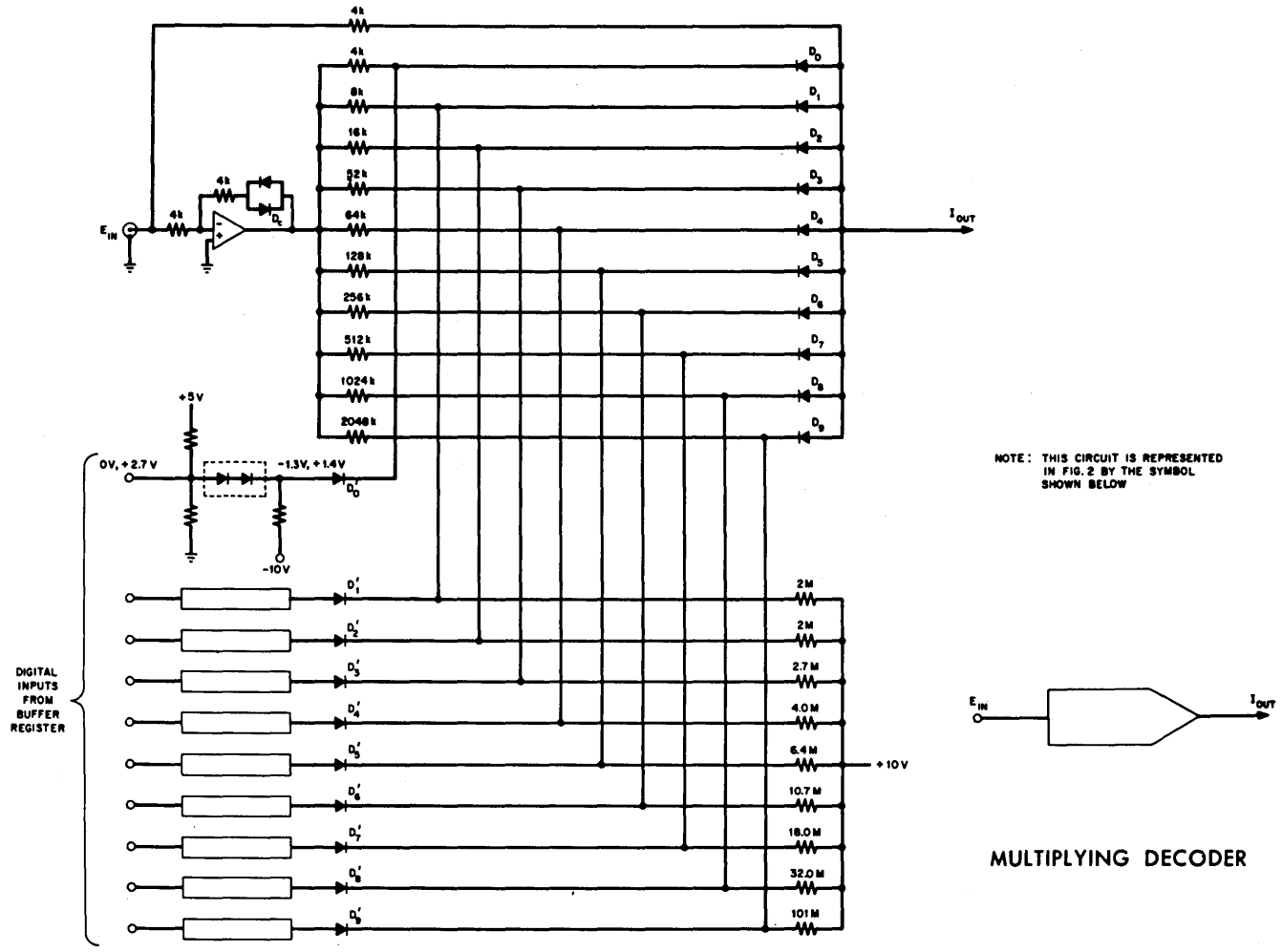


Figure 6—Multiplying decoder. Schematic diagram (output amplifier not shown)

tional amplifier. The summed currents flow through the feedback resistor of the output amplifier by the usual operational amplifier action, developing the output voltage. The currents of several resistor diode networks and their associated driving amplifiers may be summed at a single output amplifier node (see Figure 2). The resistor feeding forward from the input to the summing node of the output amplifier serves to bias the output negatively, allowing outputs of both polarities:

$$E_{out} = E_{in}\left[-1 + \sum_{i=0}^{9}\left(\frac{c_i}{2_i}\right)\right] \tag{8}$$

$c_i = 1$ for an "on" bit

$= 0$ for an "off" bit

Positive input voltages are thereby multiplied over the range (−1.000, +0.999). The level shifting networks preceding the steering diodes D'_0 through D'_9 are designed to be driven from cascode type micrologic outputs. The pertinent characteristics of the operational amplifiers are given in Table 1.

TABLE 1

DC gain	86 dB
Small signal BW (unity gain inverting)	10 MHz
Slewing rate	100 V/μs
Input impedance (open loop)	0.5 MΩ
Output impedance (open loop)	0.5 KΩ
Output voltage	±10 volts
Output current	±30 mA

Each multiplying decoder is packaged on two 4″ X 4″ circuit cards, and two amplifiers occupying a single card.

Static error

Assume that all the diodes D_c and D_0 through

D_9 are identical and have the V-I characteristic $i=I_s(\exp[-v/k]-1)$ over the range of interest, and further that the amplifiers have zero output impedance and infinite input impedance. Referring to Figure 7, where the most significant bit only is shown, the diode D_o in the feedback path of the driving amplifier provides exact compensation for the voltage drop in diode D_c:

$$E_A = -(I_{in}R + V_{DC}) = -(E_{in} + V_{DC}) \quad (9)$$

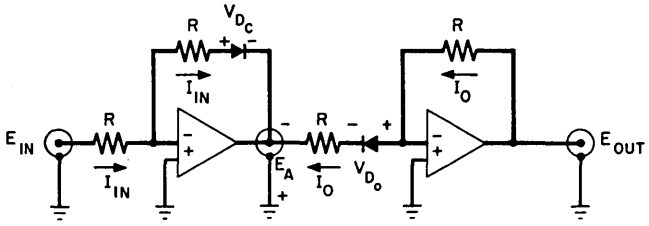


Figure 7—Simplified diagram of multiplying decoder showing most significant bit and output amplifier

But, this is precisely the voltage required to cause I_{in} to flow in D_o; hence in the feedback resistor of the output amplifier. Thus, $E_{out} = I_{in}R = E_{in}$ for all E_{in}. The diode drops in the succeeding lower order bits are only approximately cancelled out by the drop in D_c because their currents (when they are conducting) are less than I_o i.e. $\frac{I_o}{2}, \frac{I_o}{4}, \frac{I_o}{8}, \ldots$ This results in an error in the current, contributed to the summing node by all other active bits. The correct current for the i^{th} leg is:

$$I_i = \frac{E_{in}}{2^iR}$$

and the actual current $\frac{E_{in}+(i)(k)}{2^iR} = \frac{E_{in}}{2^iR} + \frac{ik}{2^iR}$

where k is the exponential factor in the diode equation. If a current source of strength $\frac{-ik}{2^iR}$ were connected to the i^{th} node, the current contribution from that bit when it is "on" would be $\frac{E_{in}}{2^iR} + \frac{ik}{2^iR} - \frac{ik}{2^iR} = \frac{E_{in}}{2^iR}$, the correct value. Note that the excess current term is independent of the input voltage so that in Figure 6, the resistors connected to the 10-volt source supply the compensating current for each node. The additional diode shunting D_c in the feedback path of the driving amplifier keeps the output from drifting into positive saturation, thereby opening the feedback loop. In normal operation it has no effect.

All computing resistors down to and including the fifth most significant bit are low temperature coefficient, thick film resistors; others are evaporated metal film. Diodes D_c, D_o, D_1, and D_2 are a quad, matched to within 3 mV over the operating current range. Diodes D_3, D_4, D_5 are 1N4153 diodes selected for low voltage drop, and the D_6, D_7, D_8, and D_9 are unselected diodes of the same type. The steering diodes D'_o through D'_9 are unselected 1N4153's. The points for the dc error curve in Figure 8 were obtained for three levels of analog input by turning on in succession each bit of a typical decoder and measuring the output voltage. The error in each case is the variation from the nominal output.

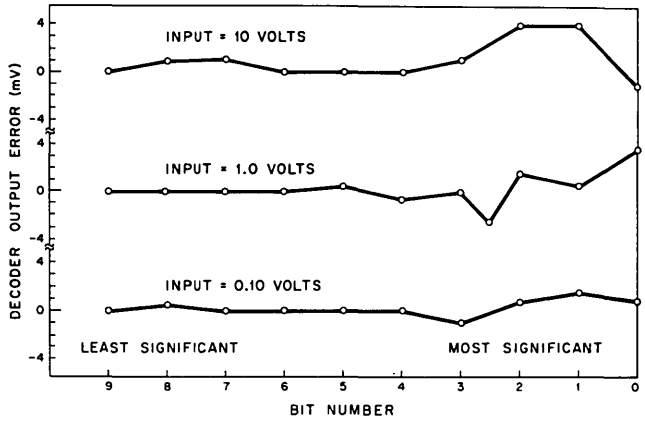


Figure 8—Static error curve of a typical decoder

The effect of the compensating current sources is shown graphically in the photographs of Figure 9. The top photograph shows the ramp output of a t decoder driven by the t counter with constant voltage applied at the analog input terminal. The bottom photograph shows the same output with the compensating sources turned off.

Dynamic characteristics

High speed accuracy of the multiplying decoder for constant digital input is limited by the bandwidth and settling time of the two operational amplifier cascade. Figure 10 shows the small signal frequency response of a multiplying decoder. The input signal was 1 volt peak to peak on a +3 volt bias. The 3db point is 4 MHz. Wide bandwidth is necessary for accuracy at high drawing rates. A high slewing rate capability is required to initiate new conditions at the start of a new curve segment. Dynamic accuracy for changing

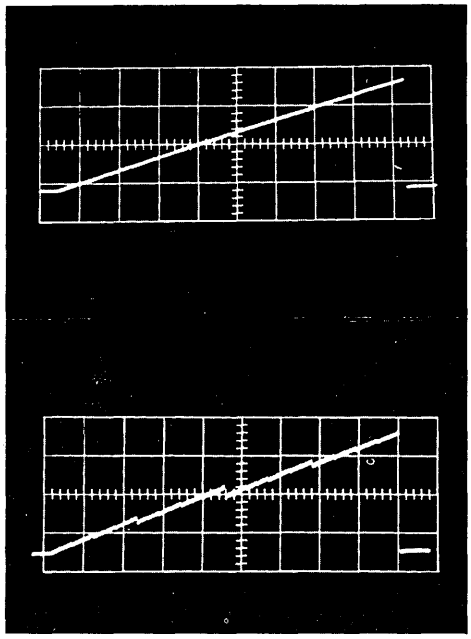


Figure 9—Ramp output of a t decoder with constant input with (a) compensating source on, and (b) compensating source off. 0.1 v/, 1ms/cm.

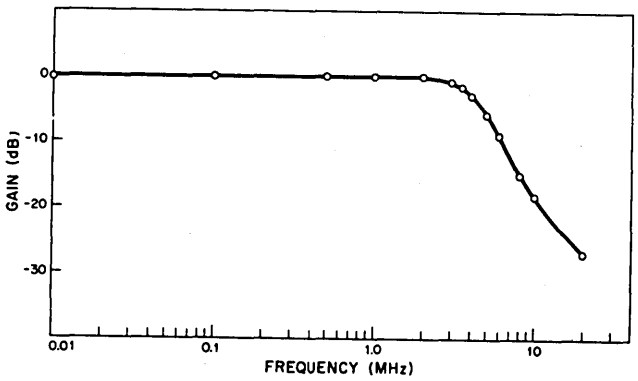


Figure 10—Frequency response of multiplying decoder.

digital inputs is limited by noise generated during switching intervals. Figure 11 shows the switching noise at the output of a t decoder for analog inputs of 1 and 8 volts when the most significant bit is switching. The noise is relatively independent of input amplitude and decreases to 0.1% of full scale in less than 1 μs. The t-decoders are identical to the other decoders. In the generator, switching noise is a limitation only in the t-decoders which switch at the counting rate. The other decoders switch once every curve segment.

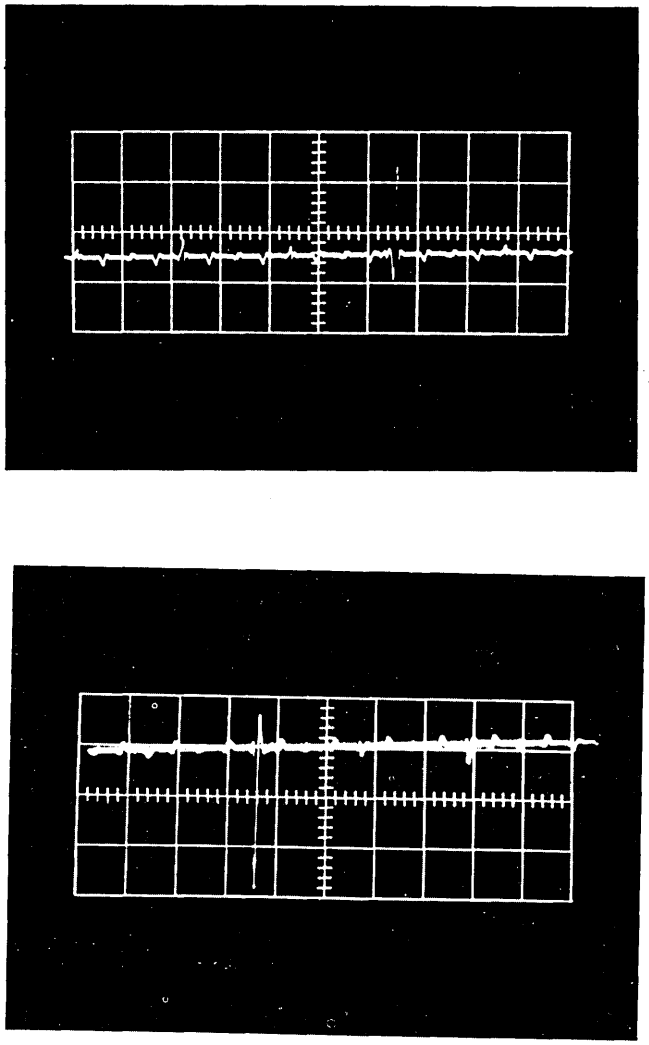


Figure 11—Output of t-decoder showing transient occurring when most significant bit switches, 0.5v/cm, 1μs/cm, (a) 1-volt analog input to decoder, and (b) 8-volt analog input.

The 4k decoder impedance level was chosen to maximize bandwidth and still keep within the output current limitations of the amplifiers. In several locations amplifier outputs were shifted from symmetrical ± 10 volt swings to 0 to -20 volt or 0 to $+20$ volt swings to increase the effective system signal level. In these cases current booster stages were also added to the amplifiers within their respective feedback loops.

Conic generator characteristics

Static accuracy

In Figure 12 the dividing loop is redrawn as an amplifier (or regulator) with linear but time varying feedback. The static error in r, that is its dc or low frequency deviation from the programmed value, depends on the loop gain in the usual manner:

$$r = \frac{S\frac{1}{w(t)}}{1+\frac{1}{1000\ w(t)}} \qquad (10)$$

The fractional error $\frac{1}{1000\ w(t)}$ is a maximum when w(t) is a minimum. Thus, over the range $0.25 \leq w(t) \leq 3$ the error varies from 0.4% to 0.03%. Generator accuracy was measured at the output terminals at dc with a DVM for $w(t) = 1$. Each of eight points over X—Y deflection field was programmed 12 different ways, i.e., using different combinations of the w's, x's, and y's giving the same nominal point with the counter stopped. The maximum deviation from the programmed value was ±0.15% of full scale.

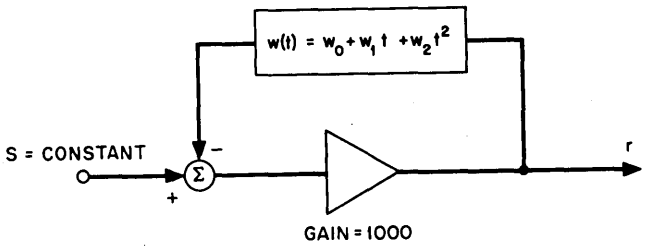


Figure 12—Conic generator dividing loop drawn as a regulator with linear but time-varying feedback.

Stability and high-speed accuracy

The large loop gains required to reduce static errors to tolerable levels would render the dividing loop unstable without frequency compensation. The single lag network shown in Figure 2 insures stability for all positive values of $w(t) \leq 3$. For $w(t) = 3$, the maximum possible value, the gain margin is 6dB and the phase margin 60°. In addition to the static error, the small signal dynamics (i.e., the ability of the dividing loop to correctly produce r(t) at high clock rates) also depend on the value of w(t) and therefore in general vary during the course of generating a segment. The dynamics of the dividing loop for any value of w(t) may be examined (at least approximately) by the artifice of stopping the counter and injecting a signal at the summing node of the t-decoder input amplifier and examining the resultant signal fed around to r. Table 2 gives the results of such a study for a step input. The responses vary from overdamped with risetime 2 μs for $w = 0.25$ to underdamped with risetime 250 ns, natural frequency 2 MHz, and damping constant 0.3 at $w = 3$. The risetimes are a measure of the delay the signal r will experience. For the smaller values of w the single lag network dominates the response. This is essentially a "velocity" error in contrast to the "position" error discussed above in the section on static error. The response of the loop in drawing a curve where there is a substantial variation in w(t) may be approximated by several piecewise calculations using the proper dynamics for each section. As an example, consider $w(t) = 0.25 + t + t^2$, $0 \leq t \leq 1$. If we break the time interval into three zones centered at $t = 0.2$, 0.5, and 0.8, we find that r experiences delays of 440 ns, 260 ns, and 150 ns.

Speed limitations

The curve drawing speed is presently limited by the noise generated in the t decoders during switching. At high rates the smeared out t-decoder spikes are noticeable. This limits the speed of curve segment generation to about 1 ms across the screen and 200 μs full scale for straight line segments. Since the switching noise is essentially independent of signal level, an increase in signal from its relatively low level (±10 volt decoder outputs) should result in a corresponding increase in S/N with attendent increased speed capabilities.

TABLE 2

w_0	w_1	w_2		Natural Frequency	Risetime 10%-90%	Damping Constant
1	1	1	Under Damped	2 MHz	250 ns	0.3
0.75	0.75	0.75	Under Damped	1.5 MHz	350 ns	0.6
1	1	0	Under Damped	1 MHz	350 ns	0.8
1	0	1	Under Damped	1 MHz	350 ns	0.7
0	1	1	Under Damped	1 MHz	350 ns	0.7
0	1	0	Critically Damped		600 ns	
0	0.5	0	Over Damped		1 μs	
0	0.25	0	Over Damped		2 μs	

SUMMARY

The present limit on drawing speed is set by the t-decoder switching noise rather than the bandwidth of the dividing loop and multipliers external to the loop. It should be possible to raise the limit almost in proportion to any increase in signal level, provided that decoder amplifier bandwidth is not seriously reduced in going to a higher signal level. Extension to cubics using the same techniques should be straightforward, although the additional phase lag introduced by the t^3 decoder would probably result in a slight decrease in

loop speed in order to maintain adequate stability margins.

The limitation of the range of w(t) due to amplifier saturation does not appear to require that large numbers of segments be pieced together to form curves, but this matter is still under study.

The limitation on accuracy imposed by the variation in w(t) in Equation 10 may be eliminated by introducing a compensating circuit to keep the loop gain constant by varying amplifier gain to track w(t).

ACKNOWLEDGMENT

The author wishes to thank Charles Seitz for his fine work in designing the digital control for the Display System of which the conic generator is a part.

REFERENCES

1 T E JOHNSON
Analog generator for real-time display of curves
Technical Report No 398 Lincoln Laboratory MIT 1965 DDC623945

2 T B CHEEK J E WARD D E THORNHILL
Operation and programming manual for ARDS-1 experimental dataphone—driven remote storage—tube display
Project MAC Memorandum MAC-M-336

3 J E WARD
Display hardware for dynamic man-machine interaction
20th Annual AFCEA Convention Washington D C June 7-9 1966

4 L G ROBERTS
Homogeneous matrix representation and manipulation of N-dimensional constructs
The Computer Display Review published by C. W. Adams Associates Inc 1 July 1966

5 L G ROBERTS
A conic display based on homogeneous coordinate mathematics
To be published in the Transactions of the PGEC (IEEE)

6 P A HURNEY JR
Combined analogue and digital computing techniques for the solution of differential equations
Proceedings of the Western Joint Computer Conference February 1956

System architecture for large-scale integration*

by H. R. BEELITZ, S. Y. LEVY, R. J. LINHARDT and H. S. MILLER
RCA Laboratories
Princeton, New Jersey

INTRODUCTION

The developing capability of the semiconductor industry to fabricate and interconnect a hundred or more logic gates on a single silicon chip promises to have a substantial impact upon the performance and reliability of today's computers. In just a decade, computer fabrication techniques have progressed from a single vacuum tube gate occupying many cubic inches in volume, to second generation discrete transistor circuitry, and to integrated circuit flat-packs in the third generation machines. Each successive generation has offered more computing power through faster circuitry and increased packing densities. Approximately 99% of the volume, even in densely packaged third generation computers, represents packaging and circuit interconnection material, and this separation between computer components still represents a severe speed bottleneck. It is not uncommon for 75% of the machine delay to occur in interconnection wiring with only 25% of the delay inherent in the flat-packs. Large-scale integration of logic gates on a single silicon chip offers promise of breaking this speed bottleneck in the larger and faster fourth generation machines.

Electrical signals must cross a multiplicity of interfaces between computer elements—bonded connections, soldered or welded connections, wire-wrap connections, and plug-in card connections. Large-scale integration offers batch fabrication of interconnections, thereby improving reliability.

Success in adapting large-scale integration (LSI) to fourth generation machines will be measured by the effectiveness of a new system organization in alleviating the highly irregular structure of current computers. These irregular structures are both inconsistent with the performance and reliability benefits offered by the LSI technology and incompatible with the practical limitations on array pins and wiring layers faced by the large-array technology. Although these latter packaging problems are not unique, they are greatly magnified with large-scale integration.

In conjunction with the Air Force Avionics Laboratory, an experimental computer capable of solving sophisticated navigational problems is being built by RCA. This computer, which has been named LIMAC (Large Integrated Monolithic Array Computer), is designed to demonstrate the LSI technology. System design concepts involve grouping of data processing structures into functional execution units that include a processing matrix and associated operand registers. Data processing control unique to the function is maintained as an integral part of the function execution unit while a simple centralized control responding to information transfer micro-instructions directs transfer of operands, data processing control words, and status words between registers over a common set of bus wires. The key aspect of the LIMAC concept is the functional partitioning of the control structures that results in the favorable use of large arrays.

The logic circuitry chosen for large-scale integration is the current mode circuitry similar to the integrated circuits used in the Spectra 70/35, 45, and 55 machines. Current mode circuits have the potential for nanosecond systems, which is consistent with the performance benefit to be derived from large-scale integration. Arrays of complementary MOS circuitry are used in an 100 nsec LIMAC memory.[1-3] Fabrication techniques are being improved to the point where reasonable yields of 100% perfect arrays will be realized.

This paper describes an approach to machine organization that leads to a simplified, and thus more regular, control structure.

*The research reported in this paper was jointly supported by the Air Force Avionics Laboratory, Air Force Systems Command, Wright-Patterson Air Force Base, Ohio, under contract No. AF33(615)3491, and by RCA Laboratories.

Current systems—partitioning problem

Historically, machine designers have encountered packaging or partitioning problems that arose when logic gates were assigned to a particular chassis or plug-in card. The partitioning objectives were to maximize packing density and to minimize the number of different card types. However, these objectives tend to be inconsistent.

The micro-programmed EDSAC machine assembled a single bit of each machine register, processing matrix, etc., on a chasis. Treatment of irregular end conditions and instruction sequencing still remains unresolved using this one-bit cross sectional partitioning approach.

More recently, a micro-functional approach to packaging has been evaluated by Pariser.[4] With this approach, five functionally interconnected modules (gating flip-flops, half adder, full adder, control amplifiers, and transfer trees) and one non-interconnected module are used as logical design building blocks. However, Pariser found that 46.5% of the logic did not use the pre-interconnected modules. Irregularity (complexity) in control structures has been recognized as the cause of partitioning problems.[5]

Control logic encompasses all logic not directly associated with the data path, and a sequence of control signals carries out a machine instruction. Control structures respond to unique instruction codes and processing feedback signals or status signals. These unique conditions give rise to a formless scattering of logic elements which in no sense can be considered regular or divisible. Regularity is taken to mean essentially the intuitive idea —if a gate appears to be connected in the same manner as its neighbors or roughly appears similar, then it is called regular. Thus, the data transfer paths between registers in a parallel machine are very much alike and are clearly regular, whereas a single gate hanging from the register, and detecting a unique condition clearly, could *not* be classified as regular. In fact, there is no reason to expect a control structure that responds to a continuum of unique conditions to have within itself a naturally repetitive character. In many machines, this control logic compromises 60% of the main frame logic.

The irregularity of the control structures is reflected in the excessive number of leads required to support a gate in the system, and there is the practical problem that the LSI technology is pin limited. A silicon chip ¼″ on a side may contain 100 pins. Using one hundred connections as a guide ,previously published data[6,7] indicate that at most 130 logic gates will be encompassed by a large array. The average number is 85 gates for 100 connections. However, an average of 85 gates per array is not too satisfying because LSI technologists are suggesting a potential in the neighborhood of a thousand gates per array.

Of course, once the arrays are packaged, there still remains the problem of interconnecting arrays with printed circuit board techniques. The number of layers of printed circuit wiring in today's computers is also a direct reflection of their highly irregular structure. A measure of the irregularity in the interconnection structure is the number of pins (flatpack or connector) serviced per square inch of printed wiring surface.

For example, an average platter in an RCA Spectra computer contains 4,680 plug-in card connection pins and services only 5.2 pins per square inch per wiring layer. A Spectra plug-in card averages seven 14-lead flatpacks and services 2.8 pins per square inch per wiring layer.

An array computer entails pin densities on the order of 26.5 to 44 pins per square inch for 60 to 100 pin arrays in flat-packs placed on 1.5 inch centers. Thus, five to fifteen layer printed circuit boards would be required to interconnect arrays if current architectural techniques are followed. The possibilities of five to fifteen layer wiring, particularly in carrying over onto the arrays themselves, is disconcerting.

Register machine concept—Step 1

The evolution of a new system organization that has a more favorable use of large arrays should move towards a machine structured of modules wherein each module consists of a completely self-contained processor having local storage, some simple processing logic, and all control necessary for the module to execute its function. In this way, each module contains a buffer or register interface with the outside world, some processing logic, a control matrix, and a set of flip-flops used in the sequential portion of control. Each module control sees only its own state. Thus the module control structure is reduced from its conventional counterpart and the requirements for communication outside the module are reduced correspondingly. These considerations lead to the *register machine* concept so named for the registers that serve as the interface for the individual module.

In contrast to earlier examples of partitioning for batch fabrication, a register machine is partitioned into separate, more sophisticated, *instruction execution units* (IEU), each of which performs either one machine instruction or a very similar set of machine instructions. A generalized block diagram of a register machine is shown in Figure 1. The boxes labeled *IEU* denote *instruction execution units* and a five-part machine bus serves as the communication channel between an IEU and main memory.

In the operation of a register machine, the contents

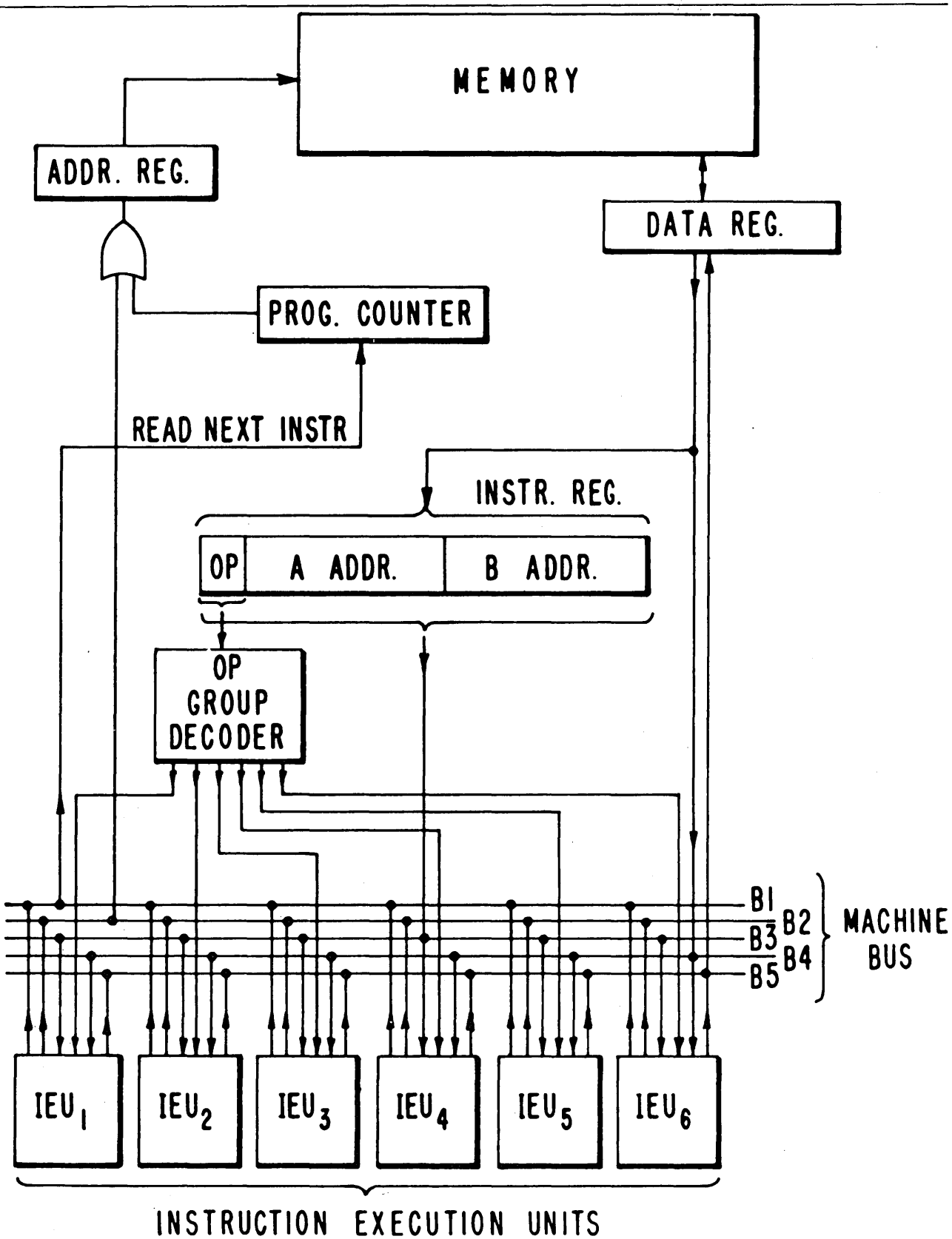


Figure 1—General block diagram of a register machine. Five parts of the machine bus are: B1-read next instruction flag; B2-memory address; B3-instruction word; B4-data word from memory; and B5-data word to memory

of the program counter are used to fetch a program instruction from memory and load the instruction register. An operation group decoder, attached to the instruction register, activates the proper IEU for execution of the Figure 2. The decoder and data processing clock together generate the proper internal sequence of enabling and inhibiting signals for the adder. A buffer or register interface provides logical interface with the IEU and memory, and each IEU contains the control logic for this register interface.

For example, an add command, fetched from memory, and loaded in the instruction register causes the *add IEU* to be enabled by the operation group decoder. Subsequently, the *add* instruction would be transferred into the register interface of the *add IEU* and the control logic for the register interface would go about fetching the two operands from memory for the addition computation. After receipt of both operands the adder logic is enabled and the operands are processed. The register interface control returns the computational result to main memory before the next machine instruction is fetched from memory.

A register machine version of a character-oriented machine (RCA 301) and a parallel word machine (RCA 4102) are described in Appendices A and B respectively. Both designs are completed down to the level of small logical blocks (e.g., 6-bit counter, 24-bit comparator, etc.) so that the NOR gate count is not precise. However, estimating the count as closely as possible but always trying to keep an error on the conservative side, the gate count in the 301 register machine is of the same order as the count in the 301 designed in the conventional manner. The distribution of the gates is changed considerably. Whereas in the conventional design some 63% of the gates are used for control, in the register machine this is reduced to less than 30%, and whereas in the conventional design the register subsystems account for only 25% of the equipment, in the register machine the register subsystems account for 65% of the machine. In the register machine, control consists of the operation decoder, the control matrices, the control counters, special comparators and address generators special decoding gates and the special control bits in the registers.

In contrast, the register machine version of the parallel word 4102 requires somewhere between three halves and wice the quantity of equipment required in the original machine. The register machine version is more regular in form, again especially in control, but the overall effect is not nearly so pronounced as it was in the case of character-oriented 301. This is a result of two factors:

(1) The 4102 data path is larger than that of the 301, and is about as regular in structure.

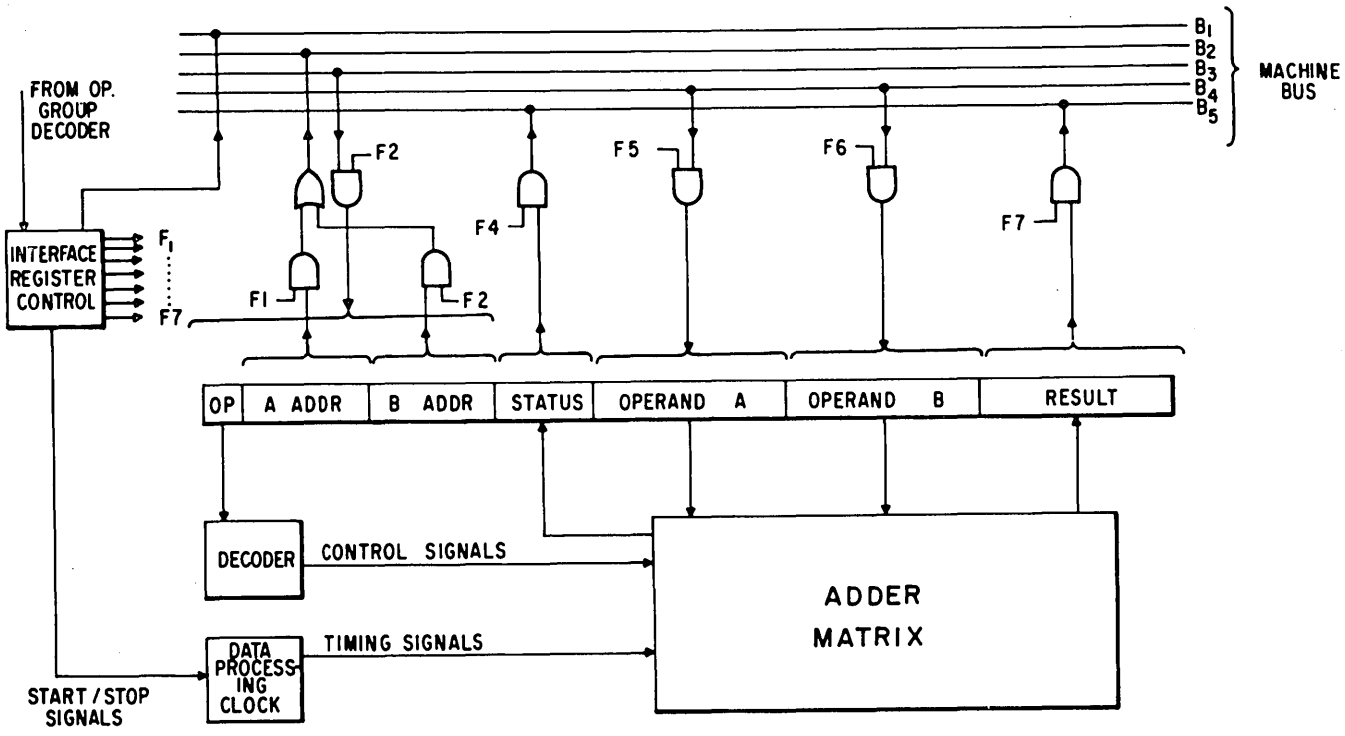


Figure 2—General block diagram of an instruction execution unit in a register machine

(2) The 4102 control is considerably smaller than that of the 301, and it is precisely in the control area that the register technique presents its greatest advantage over the conventional design methods.

Regularity, as defined in the introduction, is a property of the *gate* level of the machine. The major goal of the register machine concept is to obtain an increase in the regularity of the system. The register machine versions of the 301 and 4102 were surely an improvement over the conventional designs on this basis. The only significant candidates for non-regularity are control matrices within an IEU. Control gating and counter portions of an IEU seldom run more than 50 gates. In register machines, data paths are different in form (they are larger than in the conventional design as evidenced by the five-part machine bus) but they remain regular. The quantity of equipment devoted to control is considerably reduced and what remains may far more reasonably be called *regular* than may the control in the two conventional machines. As an added bonus, the decentralized control eliminates much of the maze of long lines which emanate from a centralized control unit.

LIMAC concept—Step 2

Functional partitioning at a machine command level was the first step in developing a system organization better suited to large arrays. The next evolutionary step introduces a second dimension in the partitioning of control. Control is now divided into two functionally independent classes, namely *information transfer* and *data processing execution*. In oversimplified terms, memory addressing control is retained as a centralized feature while data processing execution control is decentralized throughout the processing modules now called *function execution units*. This is in contrast to the register machine concept that provides each IEU with capability for data retrieval and storage.

Data processing execution control is unique to the computation logic and its connections to the associated data registers contained within one processing module, while information transfer control is common to all processing modules. Information transfer control directs instruction retrieval, data retrieval and storage, instruction and data transfers, all over a common set of bus wires. No distinction is made between data and control words and only the *names* of the transfer source and destination denote differences. Control and status words are logically complementary; a register name serving as the destination of a control word also serves as the source of a status word.

This second dimension in functional partitioning of the control structure treats operands, instructions and data addresses, and control and status words all in the same manner. This leads naturally to the elementary operation[9] (EO) format of control (also known as microprogramming).[10,11,14] A program instruction is executed by set of control words wherein each word corresponds to a single step or elementary operation in the execution of the instruction. The high degree of flexibility inherent in EO control format eliminates special-case-hardware configurations required in register machines to execute uniquely structured machine commands. The accumulator in the register machine 4102 (see Appendix B) is one example of a special-case-hardware configuration.

Both information transfer and data processing control systems employ the EO format. Within a function execution unit, states of internal data-path transmission gates and conditioning signals for the processing matrix are indicated by the binary pattern contained within the EO register (see Figure 4). The source and destination of an information transfer over the machine bus are specified by bit patterns contained in the instruction register. With EO control formats, irregular control structures are replaced by regular memory structures; the irregularity of control is transformed into the bit patterns stored in regular memory structures.

Even though the EO control format is a step towards regularizing control structures, selection of the proper EO sequence in current machines is governed by a collection of logic gates often referred to as sequencing logic. This centralized sequencing logic is complex due to the multiplicity of demands placed upon it. Numerous feedback conditions from data path logic must be handled. For example, an EO sequence for machine MULTIPLY must contain a branch operation to handle the two possibilities for the multiplier bit. In addition, other signals denote demands of input/output equipment, etc. It is the total collection of many such *conditional* feedback signals that still gives rise to the irregular complexity of the control structures in a conventional machine.

Structural differences between LIMAC and the register machine are illustrated by comparing Figures 1, 2, 3, and 4. Note that the machine bus structure has been reduced from five-parts to two-parts. In LIMAC an *information transfer bus* carries the word transfers and a *register identification bus* signifies register names of source and destination in the information transfers. Recognition logic block labeled *R*, and associated with each function execution unit as shown in Figure 3, decodes the identification signals contained on the identification bus and connects the register to the information bus as either a sender or receiver.

Unlike the register machine which operates in machine commands, LIMAC executes a string of *informa-*

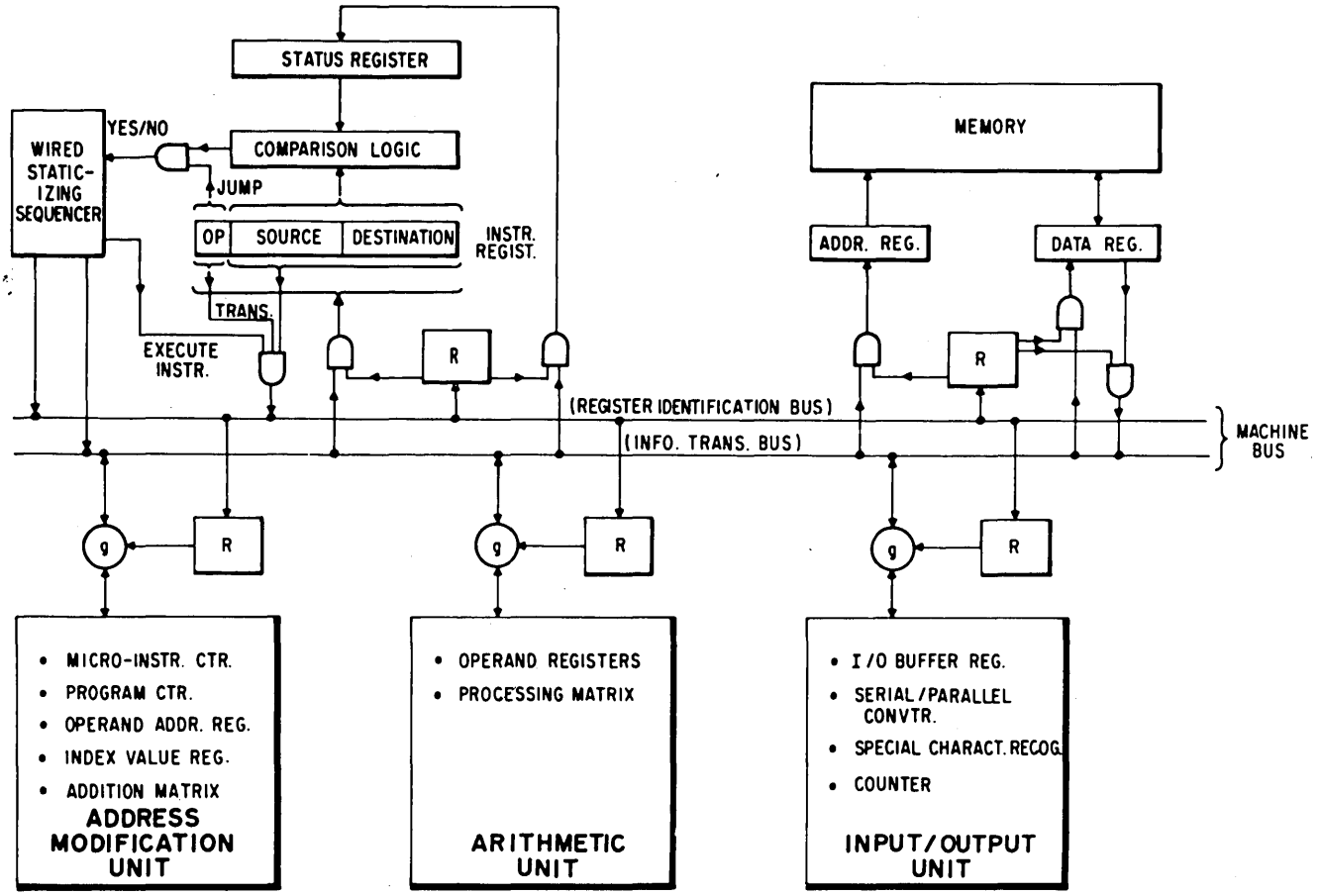


Figure 3—General block diagram of a LIMAC machine

tion transfer micro-instructions for each machine command operation code. Thus, a machine instruction is executed by first transferring operands into functional module. This is accomplished with one or more information transfer micro-instructions fetched from memory by the wired staticizing sequencer and executed in the instruction register. Another information transfer micro-instruction transfers an EO control word into the EO register of the functional module thereby specifying the connectivity of the internal data structure for execution of the machine instruction. The operation code of the machine instruction server as the base address in memory for the sequence of micro-instructions defining the machine instruction. A termination bit on the last micro-instruction of the sequence causes the wired staticizing sequencer to fetch the next machine command from memory. Similar macro-to-micro language translation techniques are employed in other conventionally organized, contemporary machines, and as LIMAC does, a variable instruction machine language format is employed.[12,13]

In the execution of complex instructions such as multiply and divide, the LIMAC concept again takes advantage of the capabilities of the large array technology. The technology offers small, fast fixed-memories. Sequences of EO control words for multiplication, etc.; can be stored in a fast, fixed-memory-contained within a functional module. The output of the fixed-memory serves as an alternate source for the gating and conditioning signals normally provided by the EO register shown in Figure 4. Rather than transferring an EO control word describing the connectivity pattern into the EO register, a start address in the fixed memory is transferred to the function execution unit. The asynchronous feature of the machine bus permits independent operation of a module once a data process operation has been initiated. In slightly oversimplified terms, each function execution unit is a very specialized computer; all are capable of independent operation.

The information transfer control system interprets only two instructions: one instruction designates by name both the source and destination registers for an information word transfer. The second instruction designates the mask pattern used for comparison with a

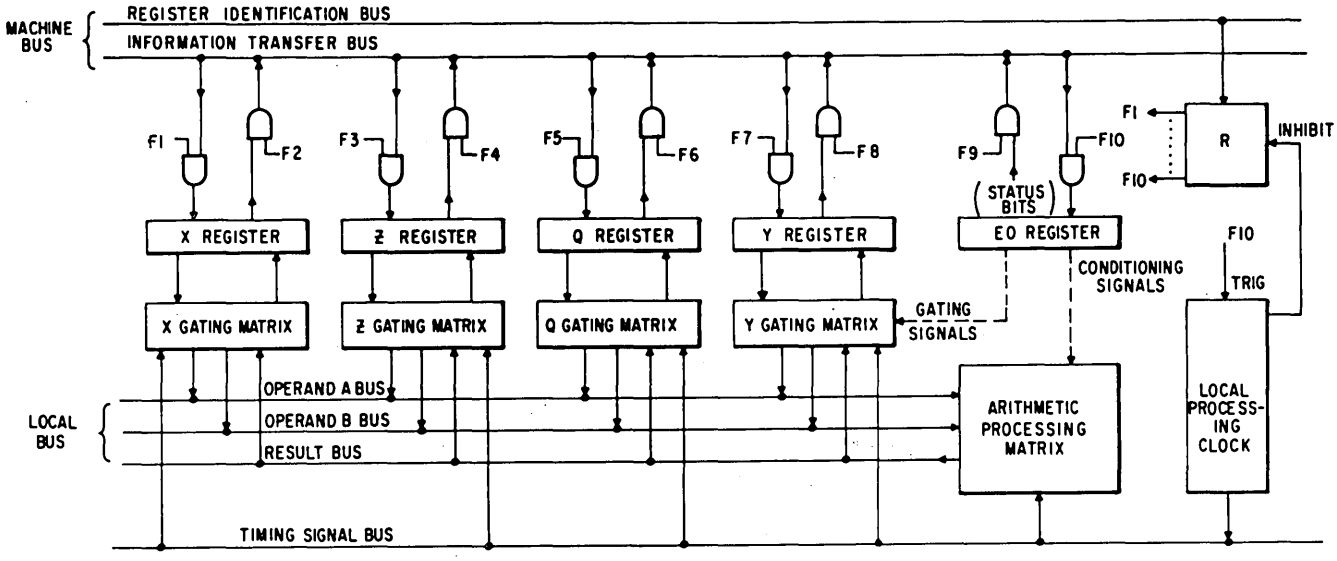


Figure 4—General block diagram of a function execution unit in a LIMAC machine

status register to implement conditional jumps. Conditional jumps are implemented by using an information transfer micro-instruction to move a status word from a function execution unit into the status register. The operation code of a conditional jump micro-instruction causes the AND/OR comparator network (see Figure 3) to match *ones* in the status register against *ones* in the instruction register. If at least one match occurs, the conditional jump is implemented by the wired staticizing sequencer. LIMAC has two fundamental types of conditional jump micro-instructions. One is called *conditional return:* if the status condition is present, software control moves to the next micro-instruction in sequence; otherwise software control is returned to the next machine instruction (macro) in sequence. The other conditional jump micro-instruction is called conditional skip: if the test condition is present software control moves to the next micro-instruction in sequence; otherwise software control skips over the next two micro-instructions and executes the third in sequence. This skip-over-two allows unlimited jumping of microprograms throughout the memory: the first micro-instruction of the two contains the information transfer instruction for loading the micro-instruction counter with the contents of the second micro-statement. The micro-instruction counter is always pointing to the next micro-instruction.

Two additional features for conditional jumps are incorporated in LIMAC. One permits reversal in responses of the wired staticizing sequencer to the Yes/No output of the comparator; i.e., the sequencer response for the presence of at least one status bit is interchanged with the response to the absence of all, and vice-versa. The second feature permits loading of the status register with the complement of the status word. Thus, the microprogrammer has four possible logical interpretations of status bits for each of the two conditional jump instructions at his disposal: (1) jump on presence of at least one bit; (2) jump on presence of all; (3) jump on absence of at least one bit; (4) jump on absence of all. The hardware cost of this added flexibility is insignificant; the savings in micro-instructions is quite significant.

Function execution unit subpartitioning

Even though the large array technology is undergoing rapid development, full integration of a complete function execution unit on a single chip may not always be possible nor advisable. Testing of exceptionally large and complex arrays and a commonality often exhibited by subpartitions of different categories of function execution units are two economic considerations that will moderate array size. A large array system organization must exhibit capability for effective subpartitioning. Figure 5 illustrates how a function execution unit is subpartitioned.

In formal terms, the composition of a functional module is a processing matrix for execution of the function and the operand registers associated with the execution of the function. Subpartitioning, then, is an iteration along the functional boundaries of the operand registers and processing matrix. Data pro-

cessing execution control is distributed among these subpartitions, and being EO formatted, distribution of data-processing control is simply segmentation of the EO register and replication of the machine bus recognition logic.

Figure 5 shows how an arithmetic unit is subdivided along the lines of operand registers with gating matrices, the arithmetic processing matrix, and the local processing clock. Each subpartition contains a small EO register that holds a subset of the control word normally stored in the EO register of Figure 4. Recognition logic for gating status bits, EO commands, and operands to and from the information transfer bus is also an integral part of each subpartition. Figure 6 illustrates the simplicity of this recognition logic. In Figure 6, the decoded signals X/S, S/R, EO/S, EO/R indicate which register (X or EO1) is to operate as a sender (S) or receiver (R) in relationship to the information transfer bus.

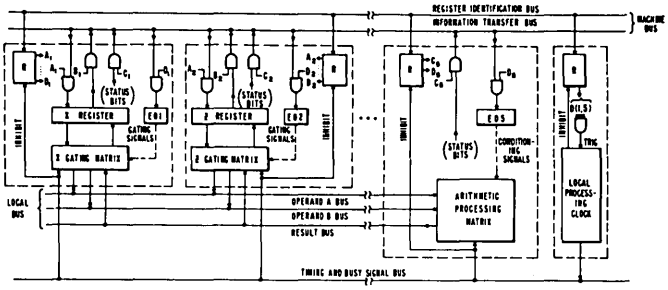


Figure 5—Subpartitioning of a function execution unit. The dashed boxes indicate the subpartitions

A *local* bus structure interconnects the subpartitions of a functional module in the same sense that the machine bus interconnects all functional modules. Local bus recognition logic is not a part of the substructures as is the case with the machine bus since the data processing execution control word is a logical union of control words EO1 through EO5.

Each function execution unit operates on its own local processing clock as indicated in Figure 5. This local clock is started by loading an EO command word into the composite of registers EO1 through EO5. Timing signals are distributed to the subpartitions on the timing bus and each individual recognition logic decoder is inhibited from operating as shown in Figure 6.

LIMAC machine instructions are executed in an asynchronous mode. Progression from one micro-instruction execution to another is controlled by reply signals; the receipt of information by the receiver register generates a reply signal which causes the wired staticizing sequencer to advance. Inhibition of either the register sender recognition logic or register receiver recognition logic by a local processing clock prevents information transfer and execution of the micro-instruction until both local data processing clocks have completed their cycles.

The utility of the LIMAC concept in terms of gate-to-pin ratios is illustrated in Figure 7. The average partitioning curve for conventional architectures[6,7] is reproduced in Figure 7 for comparison with the average for LIMAC partitioning data. Reductions in pin requirements (for reliability) by a factor of two or more are indicated. For larger arrays which would have a one-inch perimeter and support on the order of 100-pins, the average number of gates that can be utilized on the array is increased by almost an order of magnitude. LIMAC machines should benefit significantly from the performance potentials offered by the large array technology.

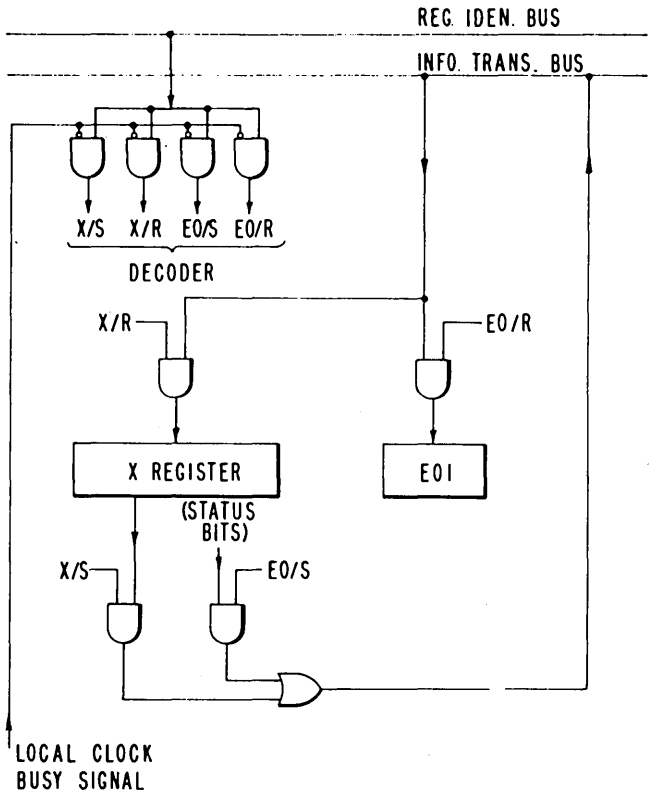


Figure 6—Recognition logic. Decoded signals X/S, X/R, EO/S, EO/R indicate which register is to function as a sender or receiver in relation to the information transfer bus

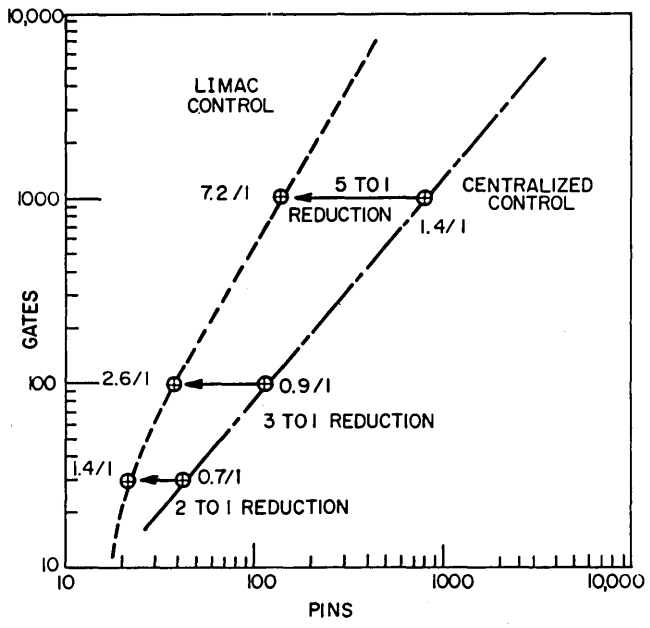


Figure 7—Comparison of the average partitioning data of conventional architectures with the average LIMAC partitioning data

LIMAC realization

LIMAC is a 16-bit, fixed-point binary machine using two's complement notation for negative numbers. The program instruction word consists of a 5-bit operation code, provisions for one index register, and a 10-bit operand address. The operation code field calls thirty-two micro-instruction subroutines that include the transcendental functions of sine-cosine, arctangent, square root as well as a standard group of arithmetic, logic, decision, and input-output editing and conversion instructions. The operand address field provides access within a 1024-word page of memory. The full memory address is formed by concatenating page bits to the operand address field. A program instruction is provided to change the page bits as desired, and up to 64 pages of memory can be accommodated using this address structure. This micro-instruction feature amounts to a hardwarized subroutine call and, in the same sense of software subroutine calls, a program instruction may pass additional information to the micro-instruction subrouting by occupying one or more successive memory locations. The executive software is coded in micro-instructions. Status words serve directly as memory addresses in the service of interrupt conditions thereby eliminating time consuming status word decoding.

LIMAC is organized into functional modules such as a complete 16-bit fixed-point arithmetic unit with a shifting and logical network, two memory units, and an input/output buffer unit. A 28-wire machine-bus serves as the intermodule communication channel, and a sequencer within the master control unit directs all communication over this channel. Memory address manipulation and program instruction decoding transformation are accomplished within the address modification unit. Each function and memory unit contains a standard control and data interface. The main memory of LIMAC is a 4096-word, 18-bit, 1μsec core array. A second 100 nanosecond memory of 256 words, 18-bits, is being fabricated from arrays of complementary MOS transistors. A standard teletype machine serves as the input/output device and is buffered by the I/O functional unit.

Arithmetic

The arithmetic unit, however, is a 20-bit parallel processor capable of both binary addition and subtraction, shifting, 12 logical and 8 counting operations. One of the extra four bits in the arithmetic unit facilitates overflow/underflow recognition while the other three are added to the least significant end of the word to improve accuracy in digit-by-digit transcendental function evaluations. Four arithmetic operand registers, Z, X, Q, and Y round out the flexibility of the unit as shown in Figure 4.

Input/Output

The Input/Output function execution unit operates as an independent I/O processor to provide a flexible man-machine interface. Although the present mechanization limits the input media to keyboard and papertape and the output media to hardcopy and papertape, the concepts are applicable to all types of input/output devices. The main function of the I/O function execution unit is to match the distinctive interface of the I/O device to the standard interface of the machine bus. In addition, the I/O module performs added functions of synchronization, word assembly, checking, and editing. In general, these miniature I/O processors can be excusively assigned to a given device or can be used to control a number of devices in a multiplex type operation. Access to main storage for transfer of data is on an interrupt basis in a manner similar to present third generation computers. In this way minimum interference with normal mode processing is obtained.

For example, consider the operation of an I/O function execution unit in controlling a single teletypewriter. In the input mode the eleven-bit teletype characters are accepted serially, assembled into machine words, and then checked for parity and control characters. Once the teletype characters have been assembled into machine words, an interrupt is initiated

and a full word is transferred to main storage. When control characters are detected, the I/O unit proceeds with the called for editing functions. One of the control characters switches the I/O unit from operating in a character mode (two characters per sixteen bit machine word) to a machine mode where characters are stripped of their higher order bits and are placed four characters to a machine word. This mode gives the keyboard operator the facility of generating machine words without code conversion. The output mode operates in a similar fashion starting with a word transfer from storage and proceeding to bit serial transmission of the teletypewriter characters. Similarly, the operational concept can be expanded to multiplex operation where the I/O function execution unit preprocesses the data from a multiplicity of teletypewriter units.

Address Modification

The address modification unit contains three address registers and one index register. One address register corresponds to the *operand address,* the second to the *program instruction address,* and the third to a *micro-instruction address.* An addition matrix modifies the operand address by the contents of index register, as well as incrementing the addresses of the program and micro-instructions. A set index flag in the program instruction automatically initiates operand address indexing.

The address structure for micro-instructions also employs paging. However, micro-instruction pages are divided into 128 word pages, and up to 512 micro-instruction pages can be addressed. Each page corresponds to a separate set of 32 micro-instruction subroutines. The 5-bit operation code of the program instruction forms the five most significant bits of the 7-bit micro-instruction page address. The full memory address is formed by amending to the operation code a 9-bit page prefix and a suffix of two zeros. The salient point is that instruction word decoding transformations are accomplished with a special composite register name serving as the destination of the information transfer staticizing instruction. Operation code decoders are not employed.

Information transfers to and from memory are made in an indirect mode. The contents of the register named as a source or destination for the memory transfer serve as the memory address. Decoders within each memory unit permit only the proper memory address register to accept the address from the machine bus. Both core and MOS memory units are equipped with address and data registers and recognition logic.

Micro-Instructions

To round out the versatility of the micro-instruction language, several modifications of the basic information transfer micro-instruction are included. One micro-instruction in the repertoire permits direct referencing of the 100 nsec MOS memory. Information transfers can be made between any member of a subset of eight machine registers that include the arithmetic and address registers and any of the 256 high speed memory locations. Another micro-instruction permits transfer of the contents of a 10-bit field in the micro-instruction register into the EO register of a selected functional module. This instruction eliminates an extra reference to memory for the function execution unit EO control word. An unconditional jump micro-instruction is also included. In this case, the contents of a 13-bit field the micro-instruction register are transferred into the micro-instruction address register within the address modification unit.

Circuits

Array circuit design for LIMAC was directed towards developing logic circuits that would meet the array requirements of small area and low power dissipation, recognizing the opportunities and constraints imposed by the array environment. A number of array unit cells have been developed that promise a high degree of logic flexibility in array design at minimum cost of implementation. A unit cell is defined as a given collection or repertoire of components from which the various circuits are constructed by appropriate metallization. The diffused chips for all arrays are identical. The functional assignment of the cell is made by intra-cell metallization. Assignment of cells into functional blocks such as a full adder, are made by inter-cell metallization.

Significant savings in power dissipation and total array chip area are realized with the cell approach. Conventionally, for example, a fixed number of logic inputs, corresponding to the maximum number expected to be used, is required for each array cell. In most cases, however, one or more inputs go unused. This represents a waste of chip area which, in arrays, is inconsistent with the precept of maximizing the functional use (performance) per array. Sharing of components overcomes this waste.

One implementation of an array unit cell stressing maximum logic flexibility permits the efficient synthesis of OR/NOR gates, set-reset flip-flops, driver and receiver array interface gates, and a number of special function gates. This variable identity cell, illustrated in Figure 8, is based upon the emitter-follower-input collector-resistor-output form of a current mode logic (CML) gate. This form of CML gate has minimum area for a non-saturating circuit and develops a 600 mV logic signal with a —4.2 V power supply. Logic swings

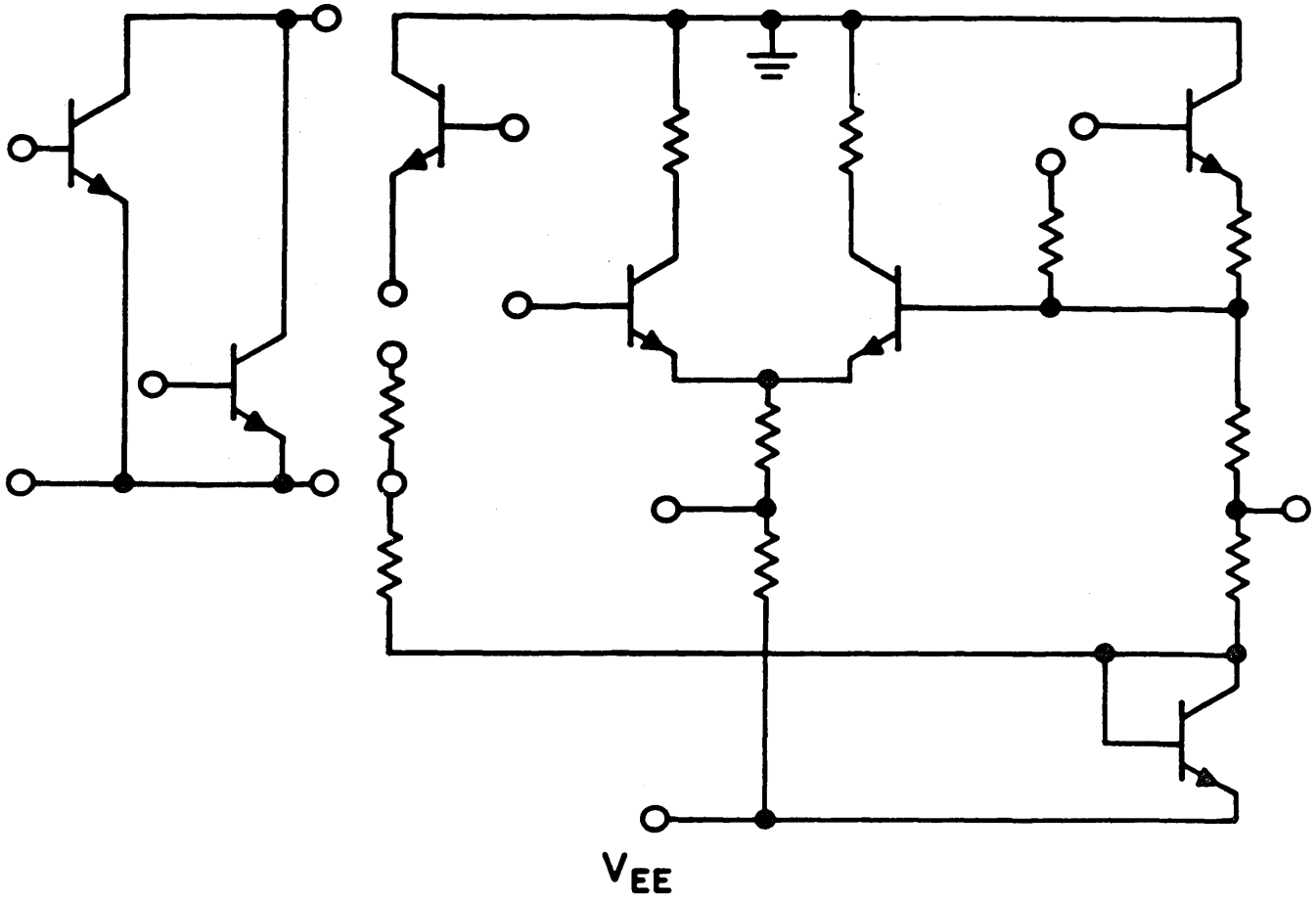


Figure 8—Variable identity array cell. Logic gates, flip-flops, and special gates for converting signal swings are formed by completing component interconnections

as low as 400 mV internal to the array are feasible. Alternate metallization permits the circuit to function at the higher logic swing (typically 800 mV) required at the array-external circuits interface to realize the required external noise margin and flat-pack CML compatibility. The power dissipation for this cell, including its built-in temperature tracking reference, is 30 mW. With a connected fan-in of four and a fan-out of six active loads plus 15 pF of wiring capacitance, pair delays of sample integrated circuits average less than 15 ns. With the circuit translated into an array environment, the stage delay reduces to 4.5 ns at a fan-out of 3. Actually, circuits utilizing this cell will be slightly faster due to somewhat smaller device area used in arrays. A number of cell layouts including dual and quad cell geometric arrangements were investigated with the goal of obtaining a compact layout. A unit cell size of 200 sq. mils area has been achieved with the quad cell arrangement using 1 and 2 mil wide resistors. With simplifications in cell design and smaller geometry resistors, a unit cell of 150 sq. mils can be obtained.

Another implementation of an array unit cell stresses minimum area and low power dissipation. This circuit is shown in Figure 9. One trial layout of this circuit, balancing minimum area against ease of wiring, resulted in a cell area of 115 sq. mils, and it is incorporated in the first LIMAC arrays. This array unit cell is also based upon the emitter follower input-collector resistor output form of CML gate but it uses a separate reference driver metallized from another cell. The reference driver provides reference voltage for up to ten array gates. Two cells are required to form a flip-flop as opposed to the single circuit flip-flop using the variable identity cell. This restriction in logic flexibility has been traded for a somewhat simpler cell configuration of fewer components and less stringent tolerances.

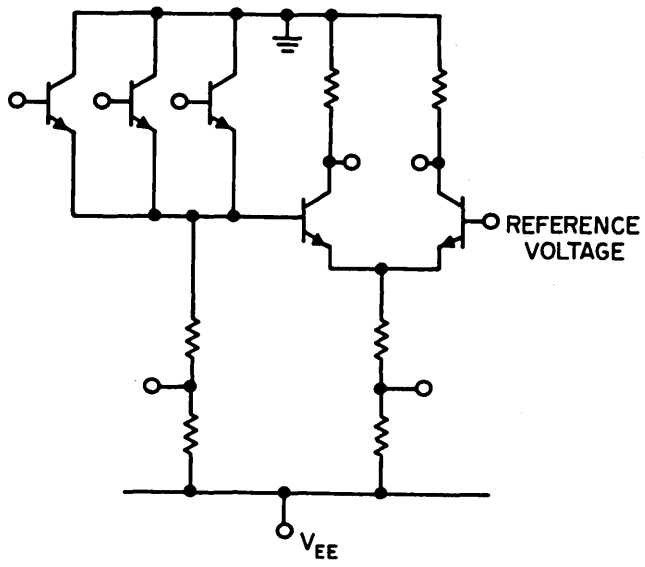


Figure 9—A small area array cell

CONCLUSION

We have described two-steps in the evolutionary development of a system architecture suited to large-scale integration. This evolutionary development has gone from conventional machine design first to a one-dimensional functional control partitioning (the register machine) and then to a two-dimensional EO formatted control partitioning (the LIMAC machine). The objective is not just to generate high gate-to-pin ratios for large arrays, but more importantly to maximize the performance benefit offered by the LSI technology through employment of a functional module organization. The second step in evolutionary development retains this functional module organization without loss in computing flexibilities inherent in mcro-programming.

Ultimately, the large array technology offers very tight packaging of upwards of a thousand gates on

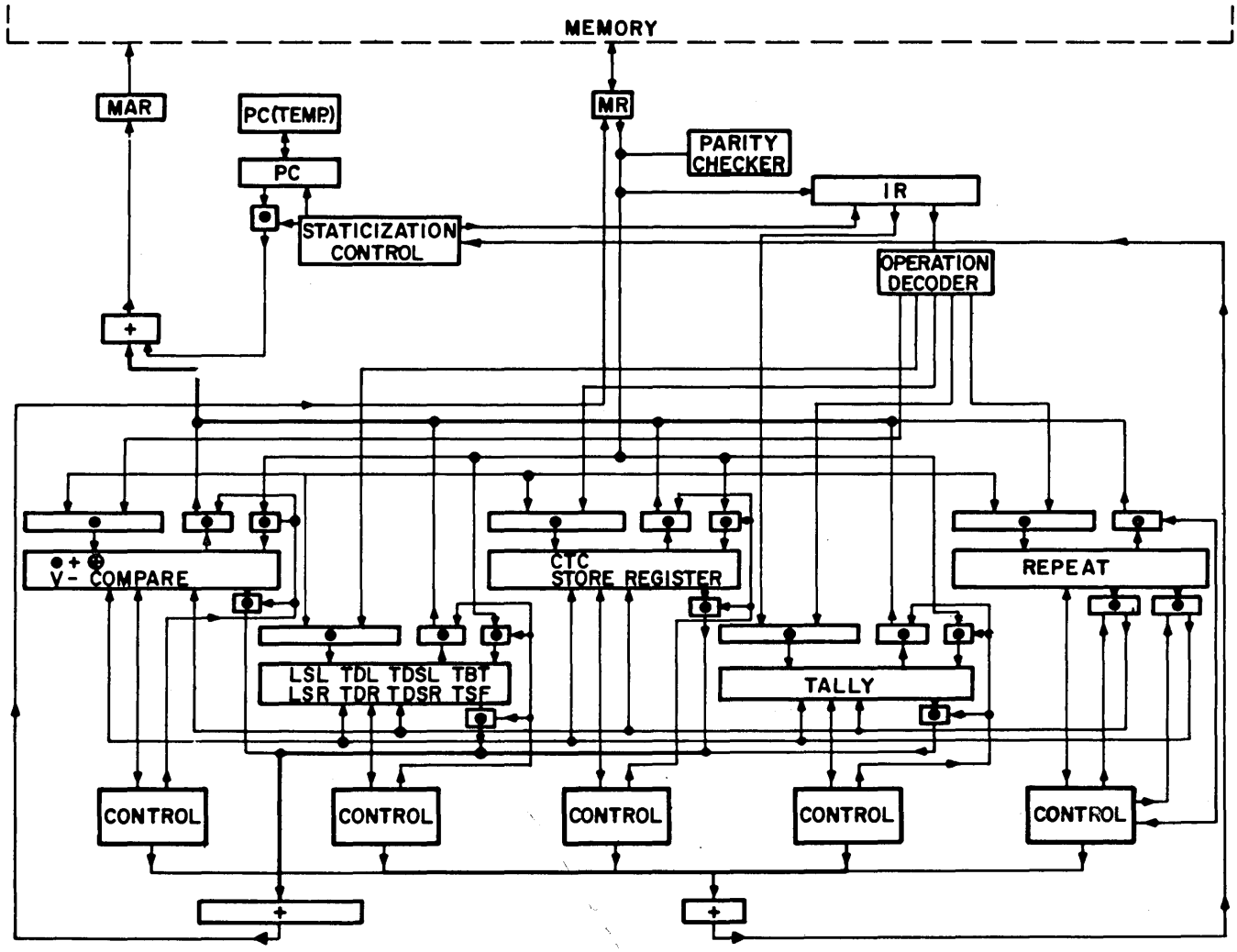


Figure A1—Block diagram of the RCA 301 as a register machine

one-quarter inch silicon chips. The technology has a practical limitation both on the number of external connections (100) to an array and the number of wiring layers available both internal and external to the arrays. The LIMAC organization is a straightforward solution to these practical problems in terms of increased system performance and reliability to be gained by applying LSI to the computer art. In fact, the LIMAC architectural concept was derived directly from the characteristics of the large array technology.

Appendix A — The RCA 301 as a Register Machine

Figure A1 is a block diagram of the RCA 301 as it would appear in register machine form. The eighteen main frame commands of the 301 were clustered into five register machine IEU's as follows:

IEU 1

And, Or, Exclusive-Or combine two variable length operands in the obvious way.

Add, Subtract constitute the arithmetic capability of the machine. They operate on variable length operands, and are executed by table look up.

Compare compares two variable length operands by binary magnitude.

IEU 2

Locate-Symbol Left/Right searches the area between two specified addresses to find a symbol different from a specified symbol.

Transfer Data Left/Right transfers a specified number of consecutive characters in memory from one area in memory to another.

Translate by Table translates a specified number of consecutive characters in memory from one code to another by use of a table in memory.

Transfer Symbol to Fill fills a specified area in high speed memory with a selected symbol.

Transfer Data by Symbol Left/Right transfers data terminated by a selected symbol from one high speed memory area to another.

IEU 3

Conditional Transfer of Control senses conditions of the machine which have resulted from various of the operations and selects between alternate sequences of instructions.

Store Register places the contents of a specified register into high speed memory.

IEU 4

Tally permits looping through a sequence of operations by automatically decrementing a stored quantity each time control returns to the start of the sequence. When the quantity is exhausted the machine exists from the loop.

IEU 5

Repeat causes all instructions between the repeat instructions and the next instruction belonging to a fixed subset of instructions, to be repeated a specified number of times.

There is also a *Halt* instruction which requires no IEU for its execution.

In clustering instructions into IEU's, our prime criterion is that the common data path equipment required of the instructions in an IEU be used in an essentially similar manner by each instruction in that IEU. Though it was not the main consideration, it turned out that within an IEU the structure of the operation flow-charts appears reasonably similar, sometimes being virtually identical as in the case of the "And" and the "Exclusive-Or" instructions, sometimes differing essentially only by a permutation as in the Transfer Data Left and Transfer Data by Symbol Left instructions. This is illustrated in Figure A2 with three instruction flow charts: Transfer Data Left, Transfer Data by Symbol Left, and Compare. Transfer Data Left transfers N characters from a set of consecutive locations in core memory to another set of N consecutive locations; Transfer Data by Symbol Left transfers consecutive characters from one part of core to another, but instead of terminating the transfer after a specified number of characters, the transfer terminates when a specified character is found in the memory. These two instructions belong to the same IEU and one can notice that the differences between the flow charts are few; the point in the loop at which the instruction starts is different, and the test for termination is different, otherwise they are identical. The other instruction flow chart, Compare, is from a different IEU and is very different from the first two charts.

Because the 301 is character oriented, it does not read an instruction word (ten characters) in a single memory read-out. A special control unit is required to handle the retrieving of an instruction work. This control operation, *Staticization,* also includes provision for indirect addressing.

The interface register for the 301 Register Machine IEU's is shown below:

A ADDRESS	B ADDRESS	OP	FSD	IND	N	DATA

The *OP* field holds the operation code to be performed. The *N* field is a character which specifies some information about the nature of the operand. Field *A* and *B* are the addresses used in execution of the instruction. The *FSD*-indicator box is peculiar to the register machine. The *F, S, D* bits specify the phase of operation of the instruction: *F* indicates *fetch* (data),

S indicates *store* (data) *D* indicates *do* (the computation). The *IND* bits store such information as operand signs, overflow, etc. The computation part of the command looks very much like it does in any machine. It is in the data path and remains unaltered. However, control is very much simpler. It consists of a counter and a control matrix or set of gates. The inputs to the matrix are the conditions of the register and counter. The outputs are signals which enable the computation and set the conditions of the register. In *all* cases control logic consists of no more than two levels of gating.

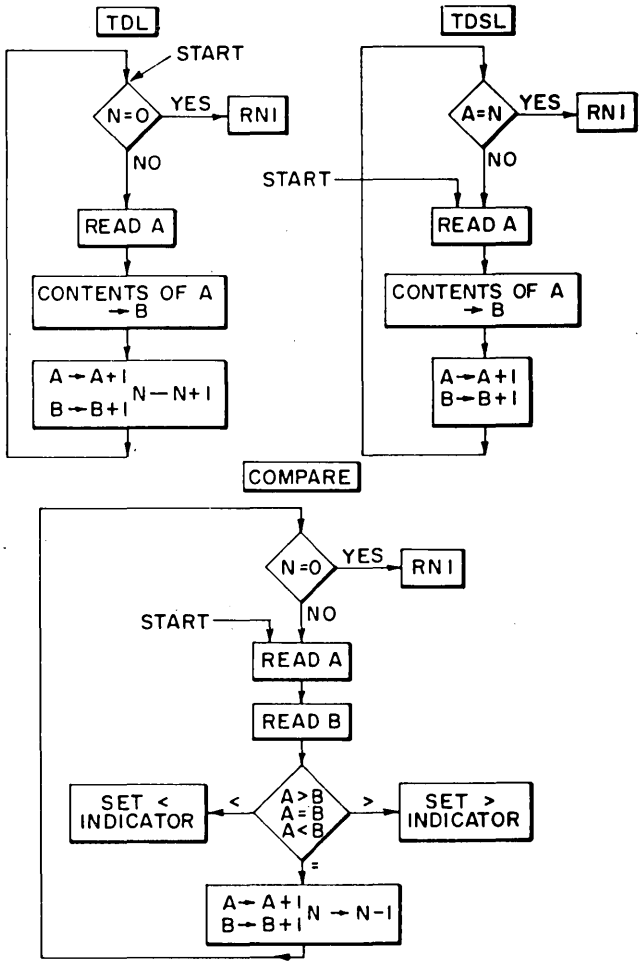


Figure A2—Representative flow charts of the 301 main frame commands Transfer Data Left (TDL), Transfer Data by Symbol Left (TDSL), and Compare. The N field represents a counter and fields A and B are addresses

The procedure for designing an IEU in the register machine, is direct. From a description of what is to be done during the command, a simple flow diagram like one used in writing a program is developed, and the transition from that to a machine control is easy.

For example consider the command TRANSFER DATA LEFT.

The instruction word is:

OP	N	A ADRESS	B ADDRESS
TDL	of CHARS TO BE TRANS	ADDRESS OF LEFTMOST CHARACTER TO BE TRANSFERRED	ADDRESS OF DESTINATION OF FIRST CHARACTER

Operation proceeds as follows:

Field *N*, the number of characters to be transferred, is examined. If N = O the instruction terminates. If N ≠ O, the character specified by the A Address is transferred to the location specified by the B Address. Fields *A* and *B* are each incremented and *N* is decremented. The process is then repeated. The flow chart shown in Figure A2 is a representation of this process. The control must do precisely what the flow chart says.

All other IEU's were designed in a similar manner. There is enough logical isolation of equipment for the design to proceed smoothly in this manner without any need to worry about a conflict of uses of any of the equipment.

APPENDIX B—The RCA 4102 as a Register Machine

The register machine design technique was also applied to a parallel, fixed word length RCA 4102. The 4102 is a small parallel computer with a full complement of index registers and arithmetic hardware. The instruction set consists of the usual data-handling and control instructions. But in addition there is a special set of secondary branch instructions which are specified simultaneously with the primary instructions and are generally used to determine the address of the next instruction on the basis of the results of the primary instruction. In addition to the complexity introduced by this secondary instruction set, operation is further complicated by the inclusion of a hardware multiprocessing capability. This involves a priority system which allows the interweaving of up to sixteen programs each with its own program counter and index registers. The programmer assigns a priority to each program, and at any given instant the program of highest priority which is in the machine is the one which is being serviced. A high priority program can relinquish control to a lower priority program, but this must be specified in the program word. As a result of this elaborate facility the execution of an instruction involves a complex sequence of operations:

(1) The instruction is extracted from memory.
(2) The primary portion of the instruction is performed.
(3) The secondary (branch) portion of the instruction is performed, and the instruction counter

associated with the active priority level is altered appropriately.

(4) The priority is checked and the instruction counter corresponding to the highest demanding priority is consulted for the address of the next instruction.

The format of an instruction word is below. The combination of fields *OP, B, L* forms a standard single address instruction, in this case the primary instruction. Fields *C* and *D* specify the branch instruction wherein the value of field *C* specifying the branch condition and the value of field *D* specifying the branch destination. Finally the bit *S* or suicide bit: If *1*, then at the termination of the instruction the active priority relinquishes the use of the machine of the next highest priority program.

The register machine technique is essentially the same as that described for the 301. However modifications in the technique are required in order to handle the secondary command and the priority level structure. In addition since a single address machine requires a register for the "other" operand, a register must be provided with access to all the instruction execution units. This register is the "accumulator." Since most instructions finish with their result in the accumulator, it is possible to perform a sequence of operations on a word without having a continual return to memory. However, in order to retain the decomposition of control, the accumulator has to exist as a passive device; it has no control of its own. Each line entering the accumulator is controlled at its source—each line leaving the accumulator is controlled at its destination.

In connection with the priority system, there is a 15-bit register (plus one implied bit), the flag register, set by program control. This register indicates those priorities which are waiting to be served. The active priority is determined strictly by the highest priority indicated in the flag register. This priority as indicated in the flag register, is used to select the appropriate instruction counter.

In the register machine version of the 4102, the index registers are set in a special scratch pad memory. All other registers perform their functions in the obvious way.

One deviation from the expected flow of operation; the indexing operation occurs during the transfer from the instruction register to the operating register; thus the operating register is always provided with the so-called effective address (E), the sum of field L and the contents of the specified index register. This is done because all commands, even those which do not access memory (like shift commands), require the effective address for their execution.

Consider the 4102 command *Add and Replace*. This command adds the contents of E, the effective address, to the contents of the accumulator and places the result of the operation in the accumulator and in E. Suppose the instruction word is:

OP	C	D	B	S	L
ADD AND REPLACE	4	3	5	0	$23221_{(8)}$

This particular command will cause the contents of [memory address $(23221)_8$ + the contents of Index register 5] (the effective address, E) to be added to the contents of the accumulator. The result will be set both in the accumulator and in E. At the termination of the Add, if the content of the accumulator is positive, the next instruction is fetched. If not, the processor goes back D = 3 steps. As in the 301, the instruction is fetched to the instruction register. From there the information is transferred both to the appropriate operating register (with indexing done en route) and to the secondary register. The primary portion of the instruction is then executed. After completion of the primary instruction, the secondary instruction is performed, and finally the flag register is consulted to determine which priority instruction counter is to be consulted for the address of the next instruction. Figure B-1 contains the overall block diagram of the machine. Notice there are five IEU's and a secondary instruction box as well as some special priority logic.

REFERENCES

1 J R BURNS J J GIBSON
Complementary MOS memories
RCA Engineer vol 13 no 3 October-November 1967

2 A H MEDWIN
Complementary MOS Technology
RCA Engineer vol 13 no 3 October-November 1967

3 P GARDNER
Fabrication of MOS memories
RCA Engineer vol 13 no 3 October-November 1967

4 T T PARISER
Connection considerations with a view toward batch fabrication
Proceedings of the National Symposium of the Impact of Batch Fabrication on Future Computers p 213 April 1965

5 J P McALLISTER
Control units in batch fabrication
Master Thesis University of Pennsylvania May 1965

6 H R BEELITZ et al
Partitioning for large-scale integration
Digest of Technical Papers 1967 Inter Solid-State Circuit Conf p 50 February 1967

7 S Y LEVY et al
System utilization of large-scale integration
IEEE Trans Elect Comp vol EC-16 no 5 October 1967

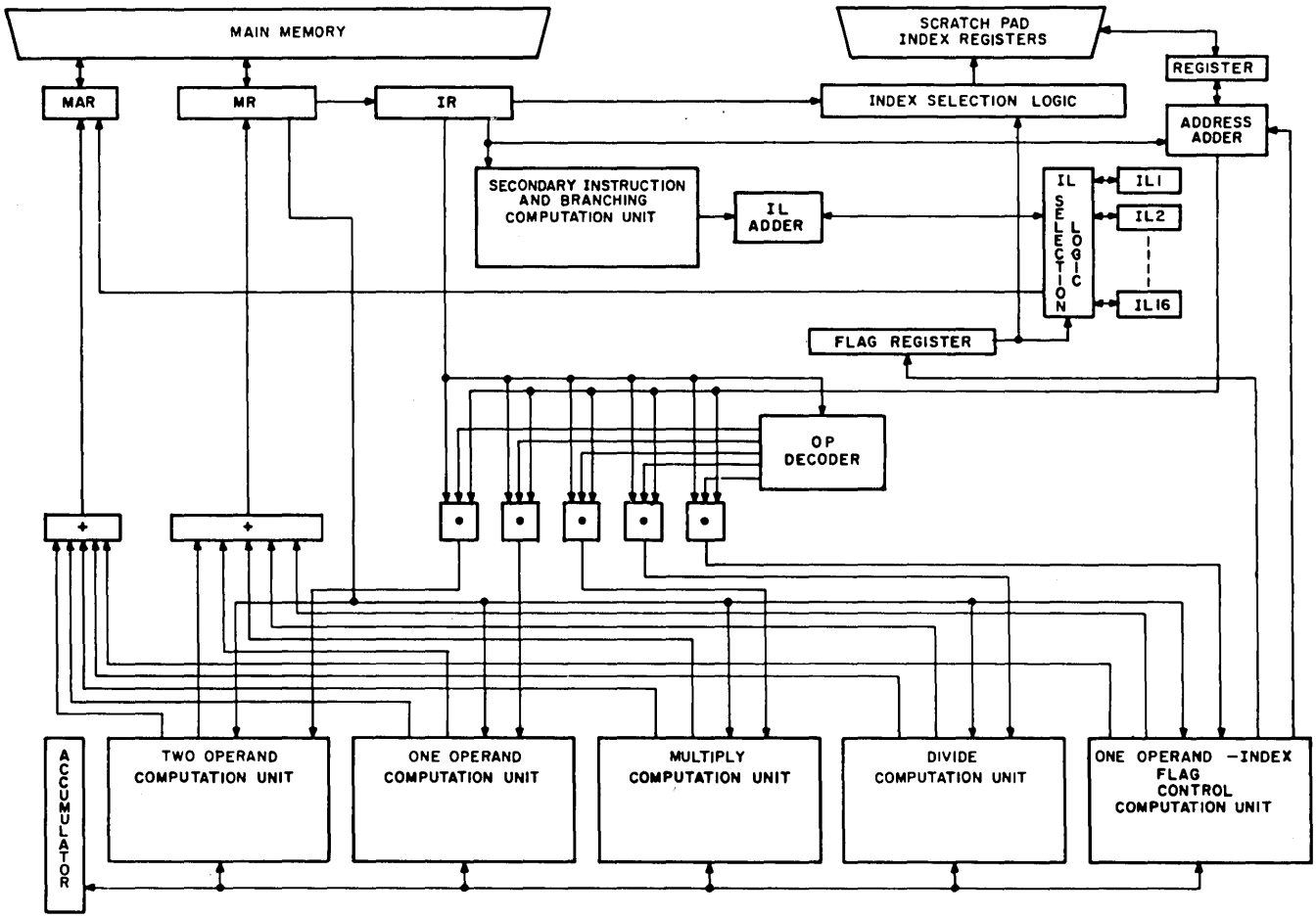


Figure B1—Block diagram of the 4102 as a register machine

8 G R GASCHNIG
Spectra 70 multilayer back panel wiring—its conception and use
RCA Engineer vol 13 no 1 June-July 1967

9 J A WEISBECKER
Data processing system
U S Patent No 3,309,679 March 14 1967

10 C R CAMPBELL D A NEILSON
Micro-programming the Spectra 70/35
Datamation vol 12 no 9 p 64 September 1966

11 M V WILKES
Micro-programming
Proc 1958 EJCC p 18 July 1959

12 A L SPENCE
Hardware and software interactions—a machine organization solution—the variable instruction computer
Presented at First Spaceborn Computer Workshop Aerospace Corporation September 1966

13 E H MILLER
RCA-USAF variable instruction computer
RCA Engineer vol 13 no 1 p 68 June-July 1967

14 M J FLYNN M D MacLAREN
Micro-programming revisited
Proceedings of the 20th Anniversary Conference of the ACM August 1967

Management of periodic operations in a real-time computation system

by HENRY WYLE and GERALD J. BURNETT
Autonetics
Anaheim, California

INTRODUCTION

I. Presentation of problems

To understand the problems of real-time control systems it is beneficial to review briefly some of the characteristics of command and control or commercial multiprogrammed systems. Typically, these systems are not time critical and as a result can schedule programs on a queue basis. The System Executive's master scheduling program keeps track of operational programs ready to run (and also keeps track of free processors if the system is a multiprocessor) and then simply assigns programs running time on a priority basis. If the processor (or processors) happens to be busy at a particular moment, the ready-to-run program list becomes a queue. Thus the timing relationships among operational programs of lesser priority are somewhat random, and there is a somewhat unpredictable wait for a given program. In many applications this random wait is acceptable; however, in many real time control systems, in particular real time avionics and space systems, this randomness is not acceptable. For example, if it is time to execute a program with a precise periodicity requirement there must be a guarantee that the processor (or a processor) is available or has lower-priority, interruptable programs in execution. If all the processors happen to be engaged in executing other programs with precise periodicity requirements at this time, a system bottleneck would exist. This bottleneck would introduce errors in the accuracies of the computations. For example, in an avionics system, if the periodicity is not precisely held, a weapon delivery program could easily cause the weapons to miss the target, or an automatic terrain-following program could cause a plane to crash or be forced to pull up to higher altitudes.

In the past, periodic programs were generally scheduled by using a real-time clock interrupt to call the highest rate periodic program in a system. The succeeding periodic programs were then brought into execution by linking them to other periodic programs with control words. This approach typically required that all the programs take a fixed amount of execution time regardless of internal branches and that almost all but the highest rate periodic program be run at rates that are higher than necessary. This inefficient use of the processor (running programs faster than necessary) has been lessened in more recent systems by varying the linkages depending on the real-time clock interrupts (sequencing a number of different programs in the time periods between interrupts). The number of interrupts can be counted by an executive program and groups of internally linked programs executed periodically every n^{th} interrupt. However, time is still wasted monitoring the clock and waiting for a program's execution time to occur. A more important inadequacy of these latter program schedulers is that they are inflexible to program changes. In particular, if a number of program changes occur, the programmer must rework many of the linkages, initializations, and time constraints by hand.

A scheduler was developed to guarantee the availability of a processor (the processor) whenever a periodic program is to be executed, to save the processor time that was wasted by the linkage scheduling method, and to provide ease of automatically handling program changes. This scheduler, described in Section II, uses a real-time clock, an ordered table of the periodic programs, and a subroutine to execute the periodic programs at their precise rates.

The subroutine (or the scheduling algorithm) is called the Local Scheduler because, in a multiprocessor* system, processor-memory combinations

*This statement and those which follow apply to a multiple-computer system as well as to a multimodule multiprocessor as in Figure 1.

are set up so that each uses a Local Scheduler and a local grouping of periodic programs. A Master Scheduler also exists in this type of system (in a single-processor system the Master Scheduler is only part of the Local Scheduler) to schedule nonperiodic programs, but it would be less efficient than the Local Schedulers at carrying out the scheduling of a large number of periodic programs. A discussion of this lack of scheduling efficiency for a Master Scheduler is presented at the end of Section II.

The periodic programs mentioned have a number of real-time sensors associated with them; as a result, because of data collection and communication delays, it is generally a difficult problem to have fresh I/O data available to a periodic program as soon as it goes into execution. Typically, a decision must be made between: (1) managing these sensors in an asychronous manner by sampling and placing them in memory at some multiple of their normal rate, or (2) managing these sensors in a synchronous manner by sampling them at the proper rate by program-controlled requests. However, either of these methods present some problems. Managing the sensors in an asynchronous manner avoids letting the information become too old between processor accesses, but it means that it is necessary to handle moderate I/O rates as if they were actually high I/O rates. As a result, extra buffering hardware and timing circuitry must be added to the computation system. If the sensors are managed in a synchronous manner, a processor might spend a considerable amount of time waiting for I/O variables that are needed very early in a program (the program is loaded and then cannot begin until it receives a number of variables from sensors). This wait could easily range to many milliseconds; as a result, the wait for I/O variables could easily waste a portion of a processor's time. From the foregoing, we can see that one way to manage the I/O traffic at a moderate rate and yet waste the minimum amount of processor time waiting for the variables is to give the I/O system the facility to call I/O variables just before the associated program goes into execution. This look-ahead feature was implemented in the Local Scheduler, and is described in Section III of this paper.

Another problem, changing the periodic program sequences, exists in managing periodic programs in a multiprocessor system that must be reconfigured after module failures. (Again, a good example is a central computation system in an avionics or space system.) Such a multiprocessor system is shown in Figure 1. All modules of the same type are exactly alike so that the highest priority computations can be continued after a number of failures as long as there is one of

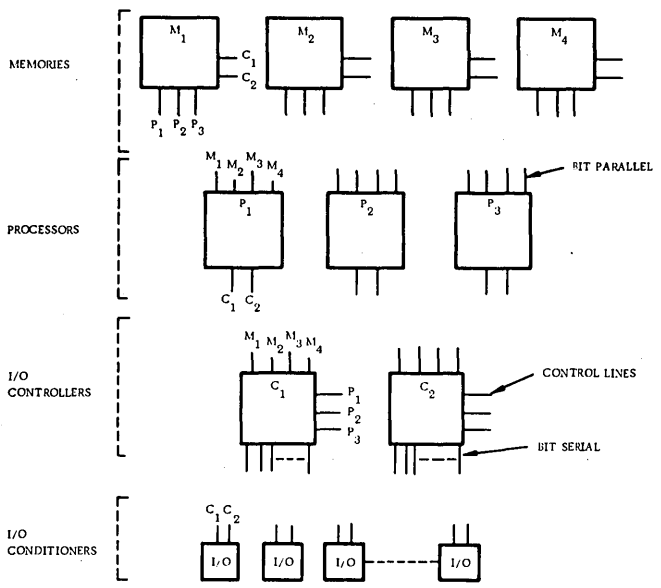


Figure 1 – Multiprocessor

each type of module present. After a failure (or failures) there is less computation power available; consequently some of the programs will be eliminated. This means that the periodic programs in a reconfigurable multiprocessor must be managed so that the setting of new program sequences is a simple task. Because the Local Scheduler was initially developed for a reconfigurable multiprocessor (shown in Figure 1), it can easily adapt to program changes. This is explained in Section II.

In summary, the Local Scheduler presented in Section II provides for guaranteed scheduling of periodic programs at their precise rates, relieves the programmer of the necessity to link programs to each other, provides for early call of I/O variables if necessary, and enables easy adapting to program sequence changes. As will be shown, this is all done with only a small overhead cost for execution of the Local Scheduler subroutine.

II. Local scheduler

As discussed in Section I, the Local Scheduler is an algorithm that uses a subroutine and a real-time clock to schedule periodic programs on a processor-memory combination. (This refers to one of the processors and memories in a multiprocessor system or to the only processor and memory in a single-processor system.) The Scheduler operates so that when the periodic programs have been completed the Executive (Central Scheduler) is notified that the processor and memory are free to do background programs.

The Local Scheduler will then interrupt the background programs and also notify the Executive when it is again time to carry out its periodic programs. The Scheduler also relieves the programmer of the task of specifying execution sequences or program linkages within the operational programs themselves. This job is not only time-consuming but also inefficient if not impossible when a large number of programs of varying periodicity must be interleaved on one processor. As mentioned earlier, a third useful property of the Local Scheduler is the ease of executing operation mode changes and of adapting to program changes either because of reconfiguration after a malfunction or because of a future system change. These points will be clarified by the following explanation of the Scheduler operation. The discussion of the early call of I/O variables is presented in Section III.

To best explain the Scheduler's operation, we will present an example. Let us assume a group of programs with periodicity requirements, as in Figure 2. Assume that the first three programs, A, B, and C,

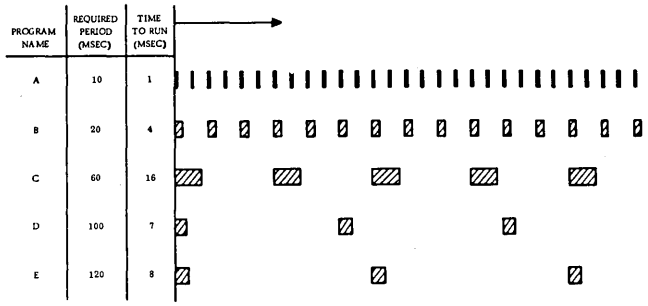


Figure 2 – A group of programs with periodicity requirements

have precise periodicity requirements and the last two programs, D and E, have only approximate periodicity requirements. The total running time of the whole program group is about 700 msec per second, so that it can theoretically be accommodated on one processor with 30 percent execution time to spare (this time can be used to work on a queue of background programs from the Master Scheduler). One possible interleaving of these programs for a single processor is shown in Figure 3. The rule used to interleave the programs is: Start out doing the most frequent program, A. Upon completion, do B, then C, D, etc. When it is time to do A again, interrupt what is being done and do A; then see if it is time to do B again, if so, do B. Keep going down the line until the interrupted program is reached and then resume it. In trying to get back to the interrupted program, several of the programs higher on the list may have to be completed. Thus, several nestings will sometimes occur. One property of the algorithm may be seen from Figure 3. Programs successively lower in the list are guaranteed precise periodic execution as long as their period is an integer multiple of the period of the program immediately above them. Whenever this rule is broken, the program breaking the rule and all programs below it can expect their execution period to be perturbed randomly, although their average execution period will remain the desired one. Thus, the chart in Figure 2 would tell us that Programs A, B, and C will have precise periodicity, and Programs D and E will be perturbed. The timing chart in Figure 3 confirms this. This property of the algorithm places no unreasonable constraint on the system. It simply says that if precise periodicity is desired the programs should be adjusted to meet the integer multiple rule.

The local scheduling algorithm is simple to implement with a real-time clock in the processor and with a subroutine. The real-time clock is set to the period of the fastest program (A). It interrupts, saves the program status, gets reinitialized, and always transfers program control to A. When A or any of the other programs is finished, the finishing program always transfers to the subroutine (the Local Scheduler). The principal function of the Scheduler is to search through a table that defines the Program Group (see Figure 4) and determine which program is next. The state of the table is shown at time $t_a = 271$ msec in Figure 4. "A" has just been completed, and the processor's real-time counter says 271; therefore, Program B will be run next. The explicit operations performed by the Scheduler Subroutine are shown in the flow chart of Figure 5.

To determine the speed of the Local Scheduler, the Scheduler was programmed using a simple instruction repertoire. Assuming a memory cycle time of 2 microseconds, the Local Scheduler consumes approximately 0.6 percent of processor time per 100 programs per second executed by a given processor. The other portion of the Local Scheduler is the real-time clock interrupt routine. It is executed at the rate of the highest rate periodic program (100/second in the case of Figure 2) and consumes about 80 usec per execution or 0.8 percent of processor time per 100 executions per second of the highest rate program. This means that the total Scheduler overhead for the example of Figure 2 is only 2 percent.

We can now specify the means whereby the scheduling operation is updated after a system reconfiguration or a change in the computation load. A system reconfiguration, discussed in Section I, is necessary in some advanced multimodule systems (see Figure 1) where it is generally desired to maintain a high relia-

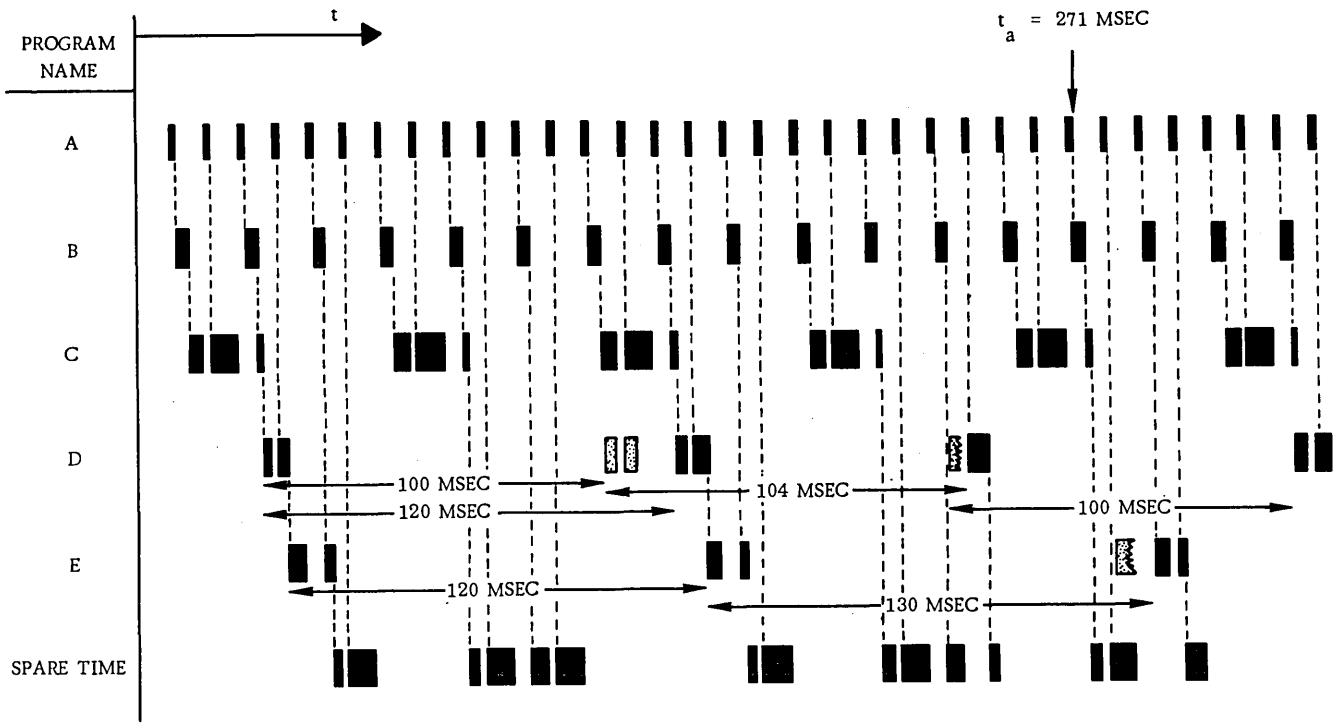


Figure 3 – A practical side for a periodic program group

REAL-TIME COUNTER: 1.00....00271

"NEXT" (NEXT TIME EXECUTION OF THIS PROGRAM IS TO BE BEGUN. SIGN BIT IS ALSO USED AS NONCOMPLETION FLAG.)	"PERIOD" (PERIOD AT WHICH PROGRAM IS TO BE EXECUTED) (MSEC)	"ENTRY" (PROGRAM STARTING LOCATION)	"RESUME" (PROGRAM RESUMPTION LOCATION)
---	10	(A) --	
1.00....00271	20	(B) 4110	--
0.00....00255	60	(C) 3170	3415
1.00....00347	100	(D) 2663	--
1.00....00285	120	(E) 5111	--

Figure 4 – Program group table of the local scheduler

bility for a certain set of computations. For example, it would be desirable to continue the navigation calculations in an avionics or space mission after a number of failures to be able to return home safely. In this case the Master Scheduler would have a table with the necessary navigation programs marked. After any failures it would be necessary to change the program sequences so that at least the navigation programs are being carried out by an active Local Scheduler.* Changes in the computation load can occur at any time in multiprocessor or single-processor systems because of new or unforeseen needs. These changes would also require changes or additions to the existing Local Scheduler program sequences. It should now be noted that both of the aforementioned operations affect the scheduling of periodic programs in the same manner, because in both cases it is necessary to set up a new program mix on the processor memory pair. The Executive Program could simply go into the Program Group Table and remove the rows devoted to the terminated programs and close up all the rows. A comparison is then made between the period of the program (or programs) to be brought in and those on the table. The new program is then placed between the two rows that bracket its period. (Alternately, bypass bits on particular table rows could be turned on and off.) The Local Scheduler is then restarted with the highest rate program executed first. It should be noted that this operation is very simple and can be carried out automatically by the

*The reader may ask why a change in program sequence would be necessary because this group of programs could have been set up together initially; however, failures early in a mission may require some portion of the navigation programs and the automatic terrain-following program to be continued in the remaining Local Scheduler. As a result, these programs may be set up together initially.

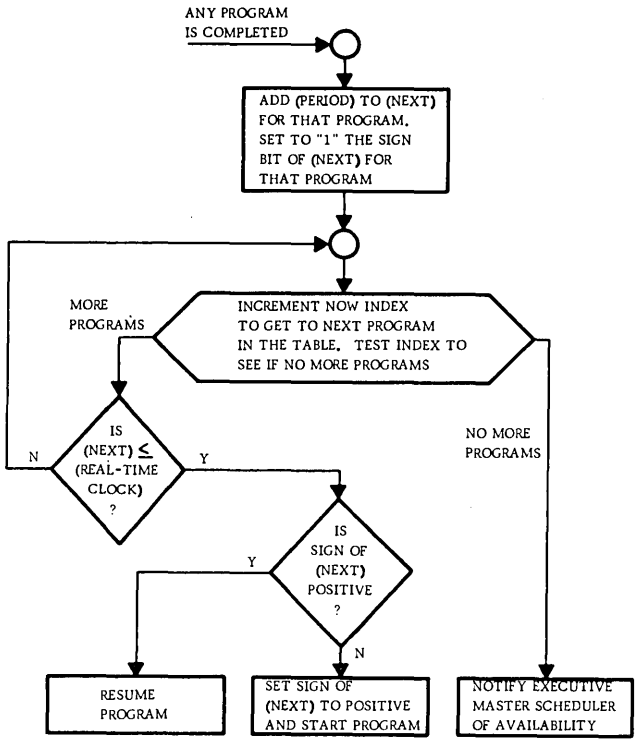


Figure 5 – Flow chart of the local scheduler

System Executive. This should be contrasted with the problems of changing linkages in a scheduling system that uses control words to link periodic programs to each other.

Having presented the Local Scheduler, it is worthwhile to briefly compare it to a Master Scheduler. Of course, a Master Scheduler is only distinct from the Local Scheduler in a multiprocessor system. In this case this large scheduler would be executed by one processor and would schedule the periodic programs for all the processors. To avoid making the system have single special modules (a single failure point), each processor module would have a real-time clock and be capable of executing the Master Scheduler; as a result the Master Scheduler saves no hardware. The Local Scheduler, as pointed out, is executed whenever a real-time clock interrupt occurs or when ever a program is completed. Therefore, the Master Scheduler processor would save the other processors the time for each to execute the real-time clock routine, but each processor would sit idle while the Master Scheduler performs its program completion and program initialization functions. Because these latter functions take the majority of the time, the Master Scheduler would increase the scheduling overhead. An even worse problem could easily arise with the Master Scheduler if two programs were to complete at once. In this case one of the two following programs would be started late. The scheduler would need to become very involved to avoid the latter problem. The need to have the Master Scheduler interruptable at all times would also inhibit the scheduling processor from carrying out precisely periodic programs. Thus the Local Scheduler is not only a very efficient scheduler for single-processor systems, but also for multiprocessor systems.

III. Early call of I/O variables

As discussed in Section I, the Local Scheduler can be used to manage I/O traffic for the periodic programs efficiently. It carries out this function by calling I/O variables for a periodic program just before it is to go into execution. In this way three important goals are achieved:

1. Separation of Computation from Input/Output
   To preserve the integrity of multiprecision numbers and of parameter sets that are internally related, it is essential to avoid computation on data in memory during the time interval in which that data is input or output.
2. Control of Transport Lags Through the Computer System
   Both uncertainties as to transport lags (minimizable by separating computations from input/output) and excessive transport lags (minimizable by establishing the proper time sequence between input, computation, and output) must be controlled.
3. Placing Input/Output Delays Off-Line from Computation
   Where input/output must take place in connection with format conversion and/or data communication, it is frequently prohibitively wasteful of computer time for the computer program to request input parameters and then halt until they arrive. Thus some sort of off-line procedure resulting in the data being available to the program when the program needs it must be used.

One possible implementation of this task is described hereinafter. This implementation may be referred to as "NESTED."

To have the Local Scheduler also carry out the early call of I/O variables, two special input timing registers, the High Rate Register and Low Rate Register, can be added to the processor. The system then operates in the same manner except for the inclusion of one new task for the Executive and another for the Scheduler.

The task for the Executive involves initially setting each processor's High Rate Register to a time delay appropriate to the sensors associated with the

highest execution-rate program of a particular program group, e.g., 500 usec. Alternatively, this register could be eliminated by the choice of an appropriate, fixed time delay. When the real-time counter in a processor has decremented to the value contained in its High Rate Register, a high rate request is automatically sent to the proper controller. The controller accepts the request (as soon as possible), goes to a preset memory position, and picks up and executes the high rate I/O program. In this way, the processor is able to immediately begin work on the high rate program when the system is interrupted for this purpose.

The new task for the Scheduler, to enable it to help call I/O variables, involves updating the Low Rate Register whenever a program is completed. The Scheduler still does its normal job of choosing the next program, but it is now also required to compare the next execution time of the program following the next program to be run and the second from top program in the Scheduler table (Program B in Figure 4). Whichever of these two programs is to be executed first has its next execution time, less some delay (say 500 usec), placed in the Low Rate Register. The Scheduler then sets up the proper memory word with the location of the I/O program for the chosen program's sensors. This is accomplished by loading an established memory position with the program starting location from the Entry column of the Local Scheduler Table, Figure 4. The controller, when requested by the low rate interrupt, goes to the established memory position and obtains and executes the I/O program. After the aforementioned extra task, the Local Scheduler as before sends the processor to the next program to be run. The explicit operation of this portion of the Scheduler is shown in the flow chart of Figure 6. This figure is a continuation of Figure 5.

The preceding I/O call routine was also programmed to determine its effect on the execution time of the Scheduler. Again, assuming a 2-microsecond memory cycle, the Local Scheduler was increased, from approximately 0.6 percent of processor time per 100 programs per second to approximately 1 percent of processor time. With the foregoing overhead, plus the 0.8 percent per 100 executions per second for the real-time clock routine, the total scheduler and I/O variable call overhead can be calculated for any system. For example, in an Avionics system under study, one processor carries out the majority of the periodic programs. This amounted to about 700 programs executions per second of which the highest rate program accounted for 200 executions per second. Therefore, this heavily loaded processor would only spend less than 9 percent of the time in overhead.

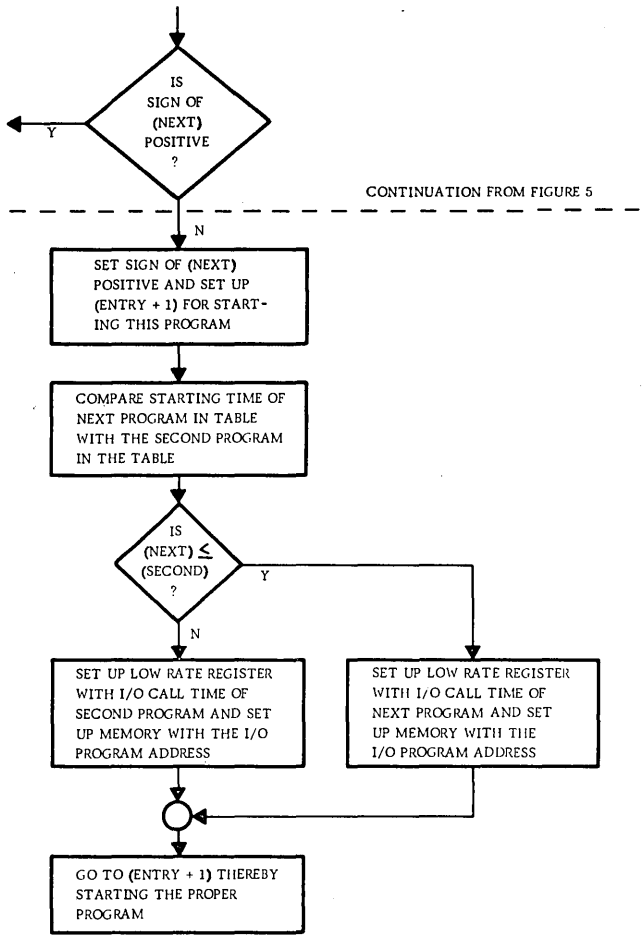


Figure 6 – Flow chart for the I/O call routine

An alternate and somewhat simpler I/O traffic management scheme can be used. This scheme, which we shall refer to as "QUANTIZED," does not require the High Rate and Low Rate Registers of the NESTED scheme, and it leads to a simpler Local Scheduler.

One difference between QUANTIZED and NESTED is that iteration rates of computations in QUANTIZED must be submultiples of the next-higher iteration rates. Thus, in Figure 2, Program D would have to be scheduled with a period of either 60 or 120 milliseconds, rather than 100 milliseconds.

A second difference between the two schemes relates to the granularity of starting times for I/O actions. In NESTED, computations can start at irregular times. The sensor inputs to these computations can be initiated (by the High and Low Rate Registers) at more or less arbitrary times prior to the scheduled computation starting times. Because the computation starting times need not be correlated with each other, neither will be the various I/O starting times. This can result in a controller executing I/O sequences nested within other I/O sequences, and can also result

in uncertainty as to the starting time of some I/O sequences. Such uncertainties of I/O execution diminish the efficiency and speed of I/O execution.

In QUANTIZED, I/O sequences can be started only upon a processor real-time counter interrupt. Furthermore, because the iteration rate submultiple method for computations leads to an easily predictable computation sequence (especially if binary submultiples are used), a corresponding predictability is associated with I/O sequence execution times. This is related to the elimination of nesting of I/O sequences in other sequences. The most practical technique for QUANTIZED has been found to be the following:

1. Group together I/O actions for computations of the same iteration rate into one long I/O sequence, or into one long Input sequence and another long Output sequence.
2. Manually calculate the maximum possible execution times of these sequences. Have the Processor's real-time counter interrupt and begin the sequences sufficiently in advance of the computation starting times that nonsimultaneity of input/output and computation is preserved. Input/Output completion timekeeping can be accomplished by a table in the Local Scheduler, by an interrupt from the I/O Controller, or both.

A possible interleaving of computations and input/output actions for the QUANTIZED scheme is shown in Figure 7. For the mix of iteration rates shown, the Processor interrupt rate (by the real-time counter) is a little more than twice the highest iteration rate of "A", the highest-rate computation

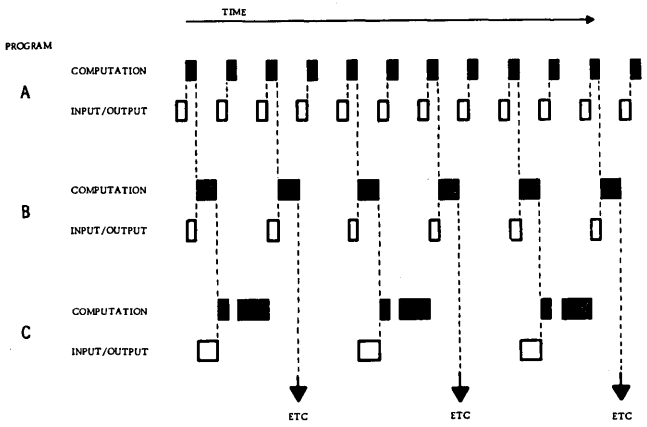


Figure 7 – A practical schedule for the QUANTIZED scheme

(By introducing still more uniformity into the time allocations for input/output sequences, a simpler pattern, more programming flexibility, and still fewer interrupts are achieved.) Several implications of Figure 7 are worth noting:

a. Input/Output and computation never occur simultaneously for computations of any given iteration rate. However, input/output for one iteration rate may occur simultaneously with computations of a different iteration rate.
b. The transport lag through the multiprocessor system (Controller input to Processor computation to Controller output) for computations of any one iteration rate is equal to the inverse of that iteration rate. This is illustrated in the detailed view shown in Figure 8. However, for a system of interlocking computations of different iteration rates, some care must be exercised to avoid transport lag uncertainties and to provide transport lags characteristic of the highest iteration rates rather than of the lowest iteration rates.

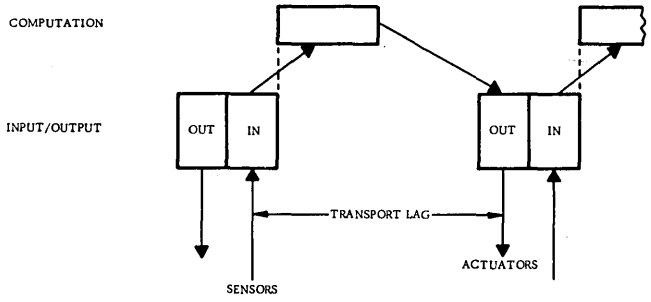


Figure 8 – Detail view of the schedule of Figure 7

Our experience has shown the QUANTIZED scheme to be of greater interest than the NESTED scheme. The QUANTIZED scheme has the further advantage of being achievable with a conventional Processor because the High Rate and Low Rate Registers are not required.

CONCLUSIONS

This paper has presented a periodic program scheduler and two schemes for controlling I/O traffic in real-time computation systems. These schemes have been shown to be very efficient in terms of use of processor time while providing the following important features:

1. Guaranteed precise scheduling of periodic programs
2. No requirement for individual programmers to set up linkages to other programs

3. Call of I/O variables just prior to program execution to limit processor waits, to provide fresh sensor data, and to prevent simultaneous computation and input/output.
4. Easy adaptation to changes in the periodic program sequences because of either the need for reconfiguration or changes in computation load

The real advantages of the complete scheduler are realized when a heavy load of precisely periodic programs must be processed. This situation is known by the authors to occur in the newer central avionics and future space computation systems; however, it may well occur in a number of other real-time control systems. In these heavily loaded systems, not only is efficiency important but also a large number of program changes generally occur in development and in the field. The Local Scheduler is easily able to adapt to these changes without loss of scheduling efficiency. As a result, the complete scheduler appears to have broad potential for application to real-time control systems.

A generalized supervisor for a time-shared operating system

by THOMAS C. WOOD
Computer Sciences Corporation
Richland, Washington

INTRODUCTION

An operating system may be considered an environment defined by a set of software processors operating in conjunction with the facilities of one or more central processors and associated devices. Its function is to allow its users to effectively command these facilities. A time-shared multiple access operating system provides apparent simultaneous availability of its facilities to a large number of users. The Supervisor of such an operating system must provide effective control of all of the facilities available to the system.

The following paper is a formalized description of the major components of the Supervisor of a time-shared operating system.

While no originality in the concepts presented is claimed, it is felt that a subject in which so much current interest is shown will benefit by such a formalization.

Supervisor

The function of the Supervisor is to provide an effective logical interface between the hardware utilized by the operating system and those modules requiring these facilities. The Supervisor directs the flow between various components of the system and resolves conflicts in the priority of usage and availability of all facilities. The Supervisor retains complete and rigid control over all input/output operations, scheduling the input/output activities to make the most efficient use of all devices, and administers all interrupts resulting from hardware activity.

The major modules of the Supervisor are:

The Interrupt Processor
The Input/Output Processor
The Timer Administrator
The Program Storage Administrator
The Facilities Administrator
The Program Administrator

Interrupt processor

The Interrupt Processor (Figure 1) may be said to be the heart of the Supervisor. It consists of two parts, the Hardware Interface and the Interrupt Activation Routine.

The Hardware Interface recognizes and responds to all physical interruptions, or traps, that the hardware is capable of generating. These traps fall into the following general categories:

I/O Traps caused by the completion, successful or otherwise, of an input/output operation. Such traps are normally generated, e.g., when a channel or device signals that it has completed a requested operation and it is free for new requests. Additionally, certain input/output devices generate attention traps which indicate the readiness of the device to receive commands.

Timer Traps caused by the overflow of a hardware clock. The clock is most often used to force periodic entrances to the Supervisor in order to prevent any program from completely dominating the facilities of the operating system.

Program Faults caused by the faulty execution of certain machine order codes. Arithmetic operations leading to register overflow or underflow are typical Program Faults. Another class of Program Faults is generated by the attempted execution of privileged or otherwise illegal instructions. The Supervisor is protected from deliberate or inadvertent interference by its clients by reserving to itself certain privileged instructions. These instructions define the storage boundaries and general operating conditions for non-Supervisor programs. An attempt either to execute these instructions or to violate the conditions results in a Program Fault. Further, in order to maintain a strictly controlled operating environment, the Supervisor has sole control over

all input/output operations. Any attempt by non-Supervisor programs to initiate input/output also results in a Program Fault.

Hardware Faults caused by hardware malfunctions. Under this heading are memory parity errors, channel and device failures and similar difficulties which may be directly detected by the hardware.

Request for Supervisor Service caused by the execution in a program of an instruction known as the Supervisor Call. Since the Supervisor allows a program to have access to none of the facilities of the hardware except the arithmetic and control units, provision must be made for programs to communicate their desires for a wider range of functions. This mechanism is the Supervisor Call. Typical requests for service are those to read and write logical records on a data set.

While the Hardware Interface responds to physical occurrences, the Interrupt Activation Routine concerns itself only with the logical consequences of these occurrences. Two queues, the Interrupt Queue, and the Priority Queue relate physical traps to logical interrupts.

The Interrupt Queue consists of one element for each logical entity for which it is desired to define an interrupt. Certain entries correspond to physical devices, e.g., a card reader; others correspond to classes of devices, e.g., tapes multiplexed on a channel; and others to conditions, e.g., Timer overflow or Request for Supervisor Service.

Each element is composed of three parts:

Device Identification indicating the logical class causing the interrupt.

Interrupt Status which specifies the nature of the interrupt.

Interrupt Director which indicates the routine, called an Interrupt Routine, responsible for servicing the logical interrupt.

By changing the Interrupt Director, the response to a logical interrupt may be dynamically altered by components of the Supervisor.

Priority among interrupts is established with the Priority Queue. The Interrupt Activation Routine scans the Priority Queue in a fixed sequence. The Priority Queue indicates the order in which logical interrupts are to be processed. By altering the priority assignments, the Supervisor may be tuned to process certain classes of interrupts more rapidly than others.

A logical interrupt is serviced by executing a series of routines beginning with that one indicated by the Interrupt Director. The interrupt is in progress, or active, until control is returned to the Interrupt Activation Routine. While only one Interrupt can be active at any time, physical traps may easily be generated at a rate faster than the Supervisor can dispose of them. These traps are mapped into their corresponding logical interrupts and stacked in the Interrupt Queue where they await servicing when the current interrupt has been completed.

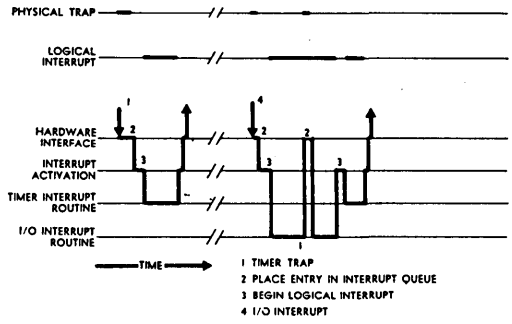


Figure 1 – Interrupt processor *typical interrupt sequence*

Input/Output processor

An Input/Output Processor is a set of routines which provides the operating system with the ability to communicate with classes of hardware devices.

The Processors schedule requests for input/output operations, initiate the physical data transfer, validate the correct transmission of data, and, when necessary, re-try operations found to be in error. The Processor is also responsible for notifying the ultimate requestor of the status of the operation upon its completion.

These activities may be grouped under the heading of:

Scheduling
Initiation
Physical Post Processing
Logical Post Processing

An Input/Output Processor generally consists of the following:

Request Queue
Schedular
Activation Routine
Input/Output Interrupt Routine
Associated Schedular Routine

Within a given time period, requests for input/output activities may be generated faster than the operations can be initiated. Requests for pending operations are placed in Request Queues by routines known as Schedulars, where they await servicing by an Activation Routine.

An entry in a Request Queue describes completely the nature of the operation to be performed. This description contains the specifications of the device

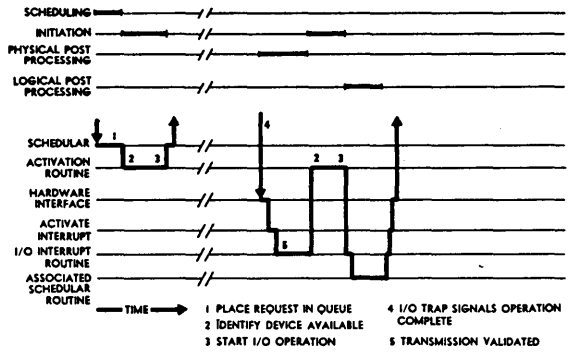


Figure 2 – Input/output processor *typical I/O sequence*

upon which the operation is to take place, the operation itself, and the identification of the routine to notify when the operation is logically complete.

The schedular is responsible for entering a request in the proper sequence into a Request Queue. Once the entry has been made in the queue, an attempt is made to start any pending operations for the class of device for which the requests are queued.

The Activation Routine scans the Request Queue, identifies devices available to service the request, and initiates all requests which may be started. When a request is started, an Interrupt Director for the logical interrupt expected is entered into the Interrupt Queue. This Interrupt Director defines the logical interrupt associated with the operation.

Upon completion of a request, an I/O trap is generated and transformed into a logical interrupt by the Hardware Interface. The operation must now be checked for correctness and the ultimate requestor, the routine which scheduled the operation, must be informed of its completion.

The Input/Output Interrupt routine specified by the Interrupt Director is responsible for validating the transmission and requesting retransmission as necessary. Once the physical operation has been accepted, the routine named in the Request Queue, known as the Associated Schedular Routine, is notified. This routine is associated with the request for operation; invoking it signals that the operation scheduled is logically, as well as physically, complete. As a part of the post processing path, pending requests are examined in the Request Queue and an attempt is made to start them. In this way the physical facilities associated with the class of request are driven at as high a rate as possible.

Timer administrator

As can be seen in Figure 3 by the interaction between Schedular, its Activation Routine, Interrupt Routine and Associated Schedular Routine, the Supervisor is basically asynchronous by nature. That is, there generally is little relationship between the order in which operations are started and that in which they terminate. Further, an operation does not have to terminate before other operations may be initiated.

Moreover, there are occasions when it is not possible to initiate an operation, but when it is desirable to be about other tasks rather than simply to wait. For this reason, the Supervisor provides itself with the ability to return to a given activity after a specified time period expires.

The Timer Administrator consists of a set of routines collectively responsible for the maintenance of physical and logical timers available for this purpose.

Logical Timers are created by the Timer Schedular and consist of entries in the Timer Queue. Each entry contains a time field and the identification of a routine, called a Timer Routine, to invoke when the time expires.

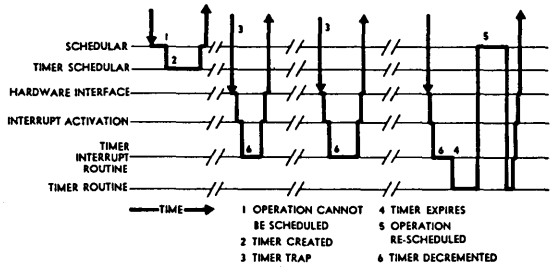


Figure 3 – Timer administrator *typical time delay sequence*

Periodically, as part of the processing of a logical timer interrupt, the timers in the queue are decremented, and upon detection of an expired Timer, control is given to a Timer Routine.

Programs

The Supervisor exists for one purpose only; to provide services to entities known as programs. A program performs some logically complete function and may not invoke or directly communicate with other programs. Communication with other programs is achieved through data sets which may be read or written by the programs. Additionally, a program is characterized by the existence of a Program Definition Area, available to the Supervisor only, which describes total environment for each program.

The primary direct service that the Supervisor provides for programs is that of Input/Output. To protect the integrity of itself, and of other programs which may be concurrently occupying Execution Storage, the Supervisor must exercise rigid control over the

total environment. Not only are programs isolated from one another by inviolable storage protect measures, but the nature and scope of Input/Output operations are limited to prevent one program from dominating the facilities of the operating system to the detriment of others.

Requests for operations forbidden to programs are made through the mechanism of the Supervisor Call, generally an instruction which generates a physical trap. Given the specification of the operation desired in the program, this specification is transformed into the corresponding action in the Supervisor as shown in Figure 4.

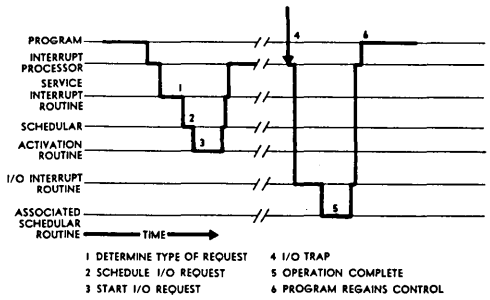


Figure 4 – Programs *typical supervisor call*

Program storage administrator

In the course of the execution of a program, the program occupies a portion of a larger space called the Program Storage. Program Storage contains the Program Definition Areas, Instruction Areas, Data Areas, and Buffers for all programs actively engaging the attention of the operating system.

That section of the Program Storage in which a program resides while making use of the arithmetic and control sections of the processor is known as Execution Storage. The remainder, that part which provides passive storage for the program, is known as Extended Storage.

At any given time, the total requirements for the Execution Storage may be expected far to exceed its availability. The Program Storage Administrator is responsible for the orderly transition of a program from Extended Storage to Execution Storage and back again.

The relationship between Extended and Execution Storage is summarized in the Execution Storage Availability Table, which relates a section, or page, of Execution Storage to that program requiring it. In the Program Definition Area of each program the Program Page Table shows for any given instant the Execution and Extended Storage assignments for the program. Programs not in a position to utilize Execution Storage are placed on Extended Storage, and the free Execution Storage pages are assigned to other programs.

In order to retain an acceptable responsiveness to programs attached to devices such as remote inquiry keyboards, the Timer Administrator periodically interrupts the execution of programs active in Execution Storage. These programs are forced onto Extended Storage while other programs are activated. This swapping process, as shown in Figure 5, insures that each program will have periodic access to the facilities of the central processor.

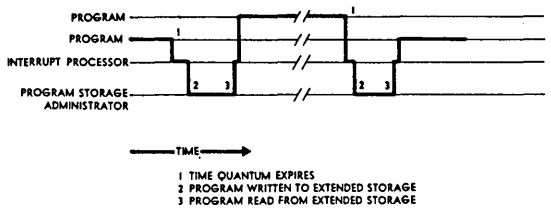


Figure 5 – Program storage administrator *typical swapping storage*

Facilities administrator

Before a program can be placed in execution, all of its requirements for facilities of the operating system must be satisfied. These facilities include catalogued data sets, such as would normally be found on bulk storage; temporary or permanent tape files; and printers or card readers. The Facilities Administrator attends to the task of satisfying requirements for given facilities. The sum of these requirements is contained in the Requirements List portion of the Program Definition Area. Once all requirements are assigned, the program may be placed in the program state.

Program administrator

The operating environment exists in one of three mutually exclusive states:

The Wait State
The Supervisor State
The Program State

The Wait State is the ground state of the operating system. It is entered whenever there is no load on the system. From this state the Supervisor State is entered whenever it is discovered that an activity has been completed or must be initiated.

It is in the Supervisor State that all interrupts are serviced, Input/Output is initiated, and response made to Supervisor Calls.

When all pending operations in the Supervisor State have been initiated, and no further processing can take place, either the Wait State is re-entered, if no program can be placed in execution, or the Program State is entered. The operating system is in the Program State as long as a program has effective control over the central processor. When this control is relinquished, either voluntarily as upon execution of a Supervisor Call, or involuntarily because of the expiration of a time quantum associated with the program or other physical trap, the system reverts to the Supervisor State. This is shown graphically in Figure 6.

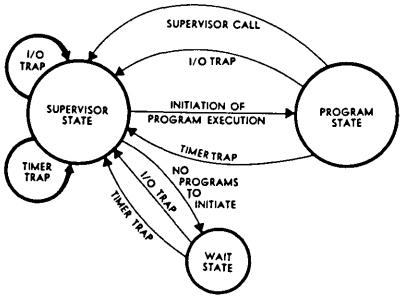


Figure 6 – Program administrator *system state diagram*

Separate from the states of the operating system are the states in which a program may find itself throughout the time allotted to it for execution. These states are governed by the attachment of the Program Definition Area to one of the following queues called, collectively, Program Queues:

Program Definition Queue
Program Queue
Execution Queue
Service Queue
Service Pending Queue
Service Delay Queue

If the Interrupt Processor is the heart of the Supervisor, the Program Administrator is the heart of the operating system, for it governs the flow between these states and controls the priorities among programs.

The Program Definition Area of a program contains descriptions of the environment of the program. The area is divided into:

The Program Descriptor which defines the number of files attached to the program, the location and extent of the program, the current location counter and accounting information.

The Machine Environment which contains the volitile machine registers and indicators. This environment is saved by the Program Administrator when the program relinquishes control of the central processor, and is restored when the program is in position to regain control.

The Requirements List which defines all data sets and physical facilities assigned to the program.

The File Control Blocks for all files attached to the program. Each file control block specifies the properties of the file and contains information relating to the current file positioning.

The Program Page Table which relates the location of pages of the program to both Execution and Extended Storage.

The Program Symbol Table which relates symbolic labels to locations within the program.

While the Program Definition Area is being constructed, and the appropriate facilities assigned to a program, the program is in the Program Definition State. (See Figure 7.) When all facilities are assigned, with the exception of Execution Storage, the program moves to the Program State. In this state the program is in contention for available Execution Storage. Once sufficient Execution Storage is made available for a program, it enters the Execution State. Programs in this state are either using the central processor or are immediately able to do so.

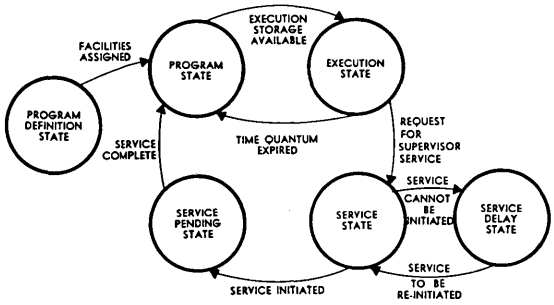


Figure 7 – Program administrator *program state diagram*

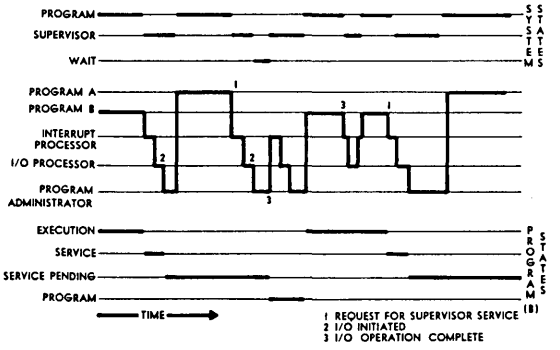


Figure 8 – Program administrator *typical program sequence*

When a program makes a request for a supervisor service, it enters the Service State where it remains until the service requested can be initiated. If the service can be initiated, the Service Pending State is entered until the service is completed. If the service cannot be started, the Service Delayed State is entered. Periodically, the program is placed in the Service State and the request re-initiated. Upon completion of the requested supervisor service, the program enters the Program State where it is again in a position to regain control of the central processor.

In order to minimize switching between the Supervisor and Program States of the operating system, all activities in the Supervisor are brought as close to completion as possible before attempting to enter the program state. If no further Supervisor activity can be attended to, a suitable program is chosen, the machine environment for this program is established, a time slice assigned to the program so that the Supervisor can ultimately regain control through a timer interrupt, and control is passed to the location counter indicated in the Program Descriptor.

SUMMARY

As systems become increasingly complex, it is increasingly important to be able to isolate the logical functions that comprise the system. The preceding paper is an attempt to specify the major functions required for the implementation of a time-shared operating system. By defining a reasonably formal structure for each function, independent of a specific implementation, a generalized machine-independent design emerges. Prototype versions utilizing this design have been implemented for two different manufacturers' machines.

A real time executive system for manned spaceflight

by J. L. JOHNSTONE
International Business Machines Corporation
Houston, Texas

INTRODUCTION

The Real Time Executive Control System discussed in this paper was the foundation for the applications programs developed in support of NASA's Gemini and early Apollo missions. Services provided by the Executive included dynamic storage management and allocation, two-level priority multiprogramming, real time data control and routing, real time error recovery, dynamic statistical monitoring, debugging facilities, and the program linkages and services that facilitated modular and independent applications system design. While a selection of these services may be available in other systems, the Executive design differs from other real time systems by these characteristics:

- Modularity—The Executive design permitted the addition of new services and facilities based on equipment changes or applications requirements with no impact on the previously provided services and facilities.
- Simplicity—Only a minimal instruction in Executive services was necessary before applications programmers could construct programs that operated in a complicated real time environment.
- Versatility—Executive could be used in the simplest simulated real time environment for the debugging of one applications program or the support of the most demanding real time missions.
- Generality—Executive was non-applications oriented; i.e., it operated equally well in a real time Gemini mission, an astronaut training session, or in a non-real time environment using simulated input from tapes.
- Invulnerable—The Executive was virtually unstoppable in real time; a feature vital for manned spaceflight.

The executive environment

The RTCC

A brief introduction to the Real Time Computer Complex (RTCC) is necessary before proceeding to any discussion of the Executive Control System. The RTCC is a functional part of the Mission Control Center at NASA's Manned Spacecraft Center in Houston, The RTCC's missions during spaceflights or training sessions are to:

- take spacecraft tracking and status data being received from NASA's global communication network and process it for display to flight controllers stationed in the Mission Control Room and the computer complex;
- compute and then forward antennae-aiming directions to tracking and communications networks all over the world so they can begin to track the manned spacecraft as it approaches;
- send calculated navigation and other information to the computer aboard the spacecraft; and
- simulate the data that network sites and space vehicles would generate during an actual mission so that personnel can be trained and equipment can be checked and readied.

To perform these missions, each of five IBM 7094-II's was assigned a different role, and the RTCC was engineered so that these roles could be exchanged at any moment. This unified set of computers allowed NASA to run either two practice missions at the same time, or a practice mission and an actual mission at the same time. Figure 1 gives a dramatic demonstration of the five systems at work in the latter configuration. In the mission configuration, network data flows into the RTCC from one of the communications processors at the Manned Spacecraft Center and is sent to the Mission Operational Computer and the Dynamic Standby Computer by a switching device called the System Selector Unit. In the simulation and training exercise, a nearby Gemini spacecraft trainer is in a closed loop system with one of the two identical Mission Operational Control Rooms (MOCR). The other MOCR is being used for the mission. One simulation computer contains a system which is generating simulated network data; the other computer is used as an operational computer. The

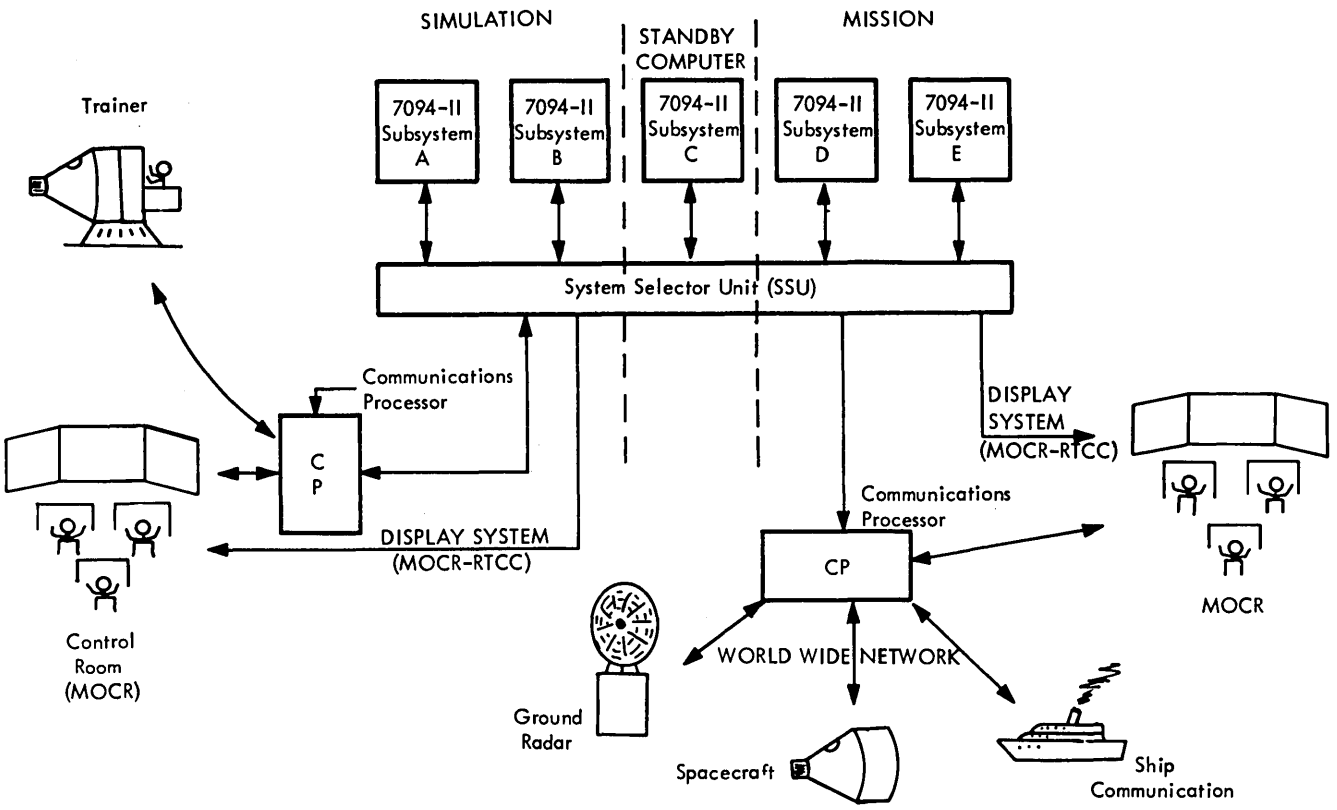


Figure 1 – RTCC data flow-simultaneous simulation and mission

fifth computer is a standby computer for both exercises; however, it is not idle but is processing a single-computer debugging exercise for checkout of a future mission or simulation system.

Although all three of the functions being performed in the RTCC are different, a single Executive is performing the control system functions for each.

The computer system

Each of the IBM 7094-II computer systems (Figure 2) in the RTCC has 65K primary memory, directly addressable through automatic relocation hardware. Each system has 524K words of Large Capacity Storage (IBM 2361) which is used as extremely high-speed buffer storage for programs and data. Programs are buffered between main memory and the Large Core Storage (sometimes termed "core file" or "COFIL") and placed in main memory wherever space is dynamically allocatable. A protect feature permits areas of storage to be protected from illegal storing operations. Tape drives are attached to standard data channels A and B. In addition, a card reader and printer are attached to channel A (not shown). The Direct Data Connection (IBM 7286) on channel C provides a rapid demand-response interface to the digital display (D/TV) television system. Access to large storage areas at a high data rate is provided by the use of the Large Capacity Storage on channel D. Real time acceptance and transmission of large amounts of data and control information are accomplished through the use of the IBM 7281-II Data Communications Channel (DCC) on channel F. At the RTCC, the DCC has 13 subchannels designated either input or output. Both the Direct Data Connection and Data communications Channel interface with the equipment and data networks serviced by the RTCC via the System Selector Unit (Figures 1 and 2).

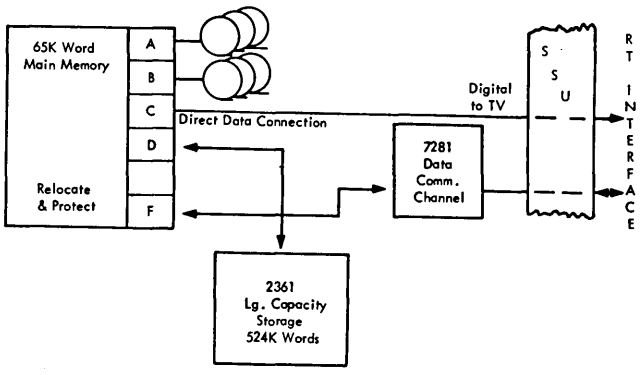


Figure 2 – RTCC 7094-II computer system

As was shown in Figure 1, the SSU permits the individual computer subsystems to be configured either singly, or in combination, to perform any of the various mission, simulation, program testing, or equipment testing functions of the RTCC. It is designed to process the inputs and outputs of up to six RTCC computers. The System Selector Unit makes switching connections between computers and their inputs and outputs and routes data accordingly. The SSU's routing assignments are made by plugboards.

The preceding paragraphs have discussed the Executive environment—the RTCC and its IBM 7094-II computer in which the Executive performed the real time computer control functions from Gemini IV to XII and Apollo 201, 202, and 203.

The real time system design

What is executive?

We have placed Executive in an environment geared to real time operation. But, to enter into any discussion of real time, one must first define *his* version of real time; for there are probably as many definitions of real time as there are real time systems. To understand the Executive real time system design, one must realize that the response time for the NASA mission application must be an increment sufficiently small to guarantee positive control of a manned spaceflight. At the RTCC, the usual time frame (or increment) in which data is received and presented to NASA Flight Controllers is considerably *less* than a second. Appreciating the response time required for the real time Executive, we can now turn to a general description of Executive.

General description

Executive is a collective term for those routines which perform the support functions for the applications programs at the RTCC. Executive has two general responsibilities in this capacity: (1) to serve as an interface between applications programs and the RTCC input/output devices and communications lines, (2) to control the execution of and communications between the application programs. Executive is a non-applications oriented system; i.e., it supports equally well all the RTCC systems: the Gemini or Apollo Mission system, the Simulation Checkout and Training System (SCATS), the Dynamic Network Data Generation system (DNDG), the Ground Support Simulation Computer system (GSSC), or the Operational Readiness and Confidence Testing system (ORACT). The application programmers who design programs for these various Executive supported systems code routines in assembly language or FORTRAN to perform mathematical computations, interpret input data, or form output data. The programmers are relatively uninformed as to how Executive works internally in performing its responsibilities. All that is required of the programmer to use Executive is a basic knowledge of the communication mechanisms with Executive and what he is to expect in the way of input from Executive.

System modularity

The majority of the application system's situations call for processing logic programs which can be segmented into controlling logic and a series of controlled processing elements. The former programs are designated supervisors, and the latter are termed processors (see Figure 3). Supervisors are multi-element programs which control processors and treat

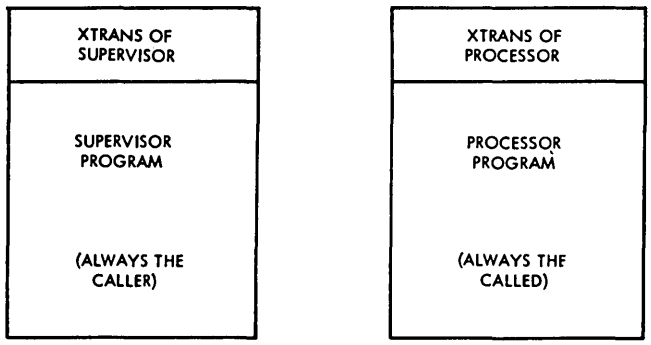


Figure 3—Supervisor and processor

them as closed subroutines. Processors differ from traditional subroutines in that no processor can call another processor. A processor can only execute and return control to a supervisor; usually, the supervisor that called it. Some processors are general in nature, as in the case of certain mathematical operations. Processors of this type can be shared by supervisors. Processors receive no input data from Executive. Supervisors receive and supply the processors with the data needed for the processor's execution.

Supervisors and processors are relocatable units; i.e., they are dynamically buffered from static storage on the Large Core Storage (LCS) to core by Executive when their logic is needed. The origin of a supervisor or processor in the 7094 core is at an address which is a multiple of 256. This address, termed base register, is set by Executive into the relocation register prior to execution of the processor or supervisor. Although every supervisor and processor is assembled with addresses relative to zero, the base address (contents of relocation register) and the offset (gen-

erated by the program at assembly time) are summed in the hardware for specification of actual memory addresses. Address protection is also performed in the hardware; the upper and lower bounds of a supervisor or processor in core are set into the protect registers by Executive when the element is brought into core. If the processor or supervisor references a memory address outside these bounds, a protection interrupt occurs. Certain "protected" instructions also cause an interrupt.

Although the Executive is primarily a core resident monitor existing in the lower 13K of the 65K core memory on the 7094, it too has about a dozen functional programs in the form of supervisors and processors that it buffers in and out of core as needed. As a comparison, the Gemini Mission System contains about twenty supervisors for centralization of flight/vehicle control logic and for supervision of data processing and mathematical computation. Nearly 250 processors are callable by the supervisors.

Standard argument area

Every supervisor or processor contains a ten-word table, called XTRANS (see Figure 3 and Figure 4.1). When any program requires processing by another program, the "calling" program fills its own XTRANS with whatever data the "called" program needs to interpret the request.

For example, in a program used to compute square roots, an XTRANS convention would be established by the program. All other programs requiring square roots would follow the convention. This convention could be: when the square root program receives control, it will calculate the square root of the quantity contained in the first cell of XTRANS. This square root program also would specify that the third word of XTRANS will always be set to zero (Figure 4.2), unless some error occurs in the square root calculations (Figure 4.3). This simple example could be complicated slightly by changing the program to a generalized root extractor. In this case, the root extractor program might define the first word of XTRANS to contain the argument, the second word to contain the power, the third to return an error code or zero, and the fourth to contain the absolute answer.[1]

Once a supervisor or processor (program) defines its input and output XTRANS, that program's services are available to any programs requiring them. When one program (usually a supervisor) calls another program (usually a processor), Executive moves the contents of the caller's XTRANS to the XTRANS of the called program. When the called program completes and returns control to Executive, Executive moves the contents of the completed program's XTRANS back into the XTRANS of the caller, as shown in Figure 5.

Standard control interface

The Executive provides a standard interface which is used to pass control between application programs (supervisors or processors). By using this interface, the Executive solves such problems as: allocating a program to main memory prior to execution, executing programs according to their priority in the system, and multiprogramming the asynchronous flow of many paths of logic. (See Multiprogramming Aspects below.)

The responsibility of determining how the Executive should pass control from program to program rests with the programmer by use of the CALL statement. The CALL statement requests a service from the Executive, while the arguments dictate how the service should be performed.

The mechanism of a CALL statement is to enter a specialized Executive routine in the resident nucleus, provide the routine with arguments supplied in the CALL statement, and have Executive execute according to the definition of the service and the supplied arguments.

To reach the resident Executive routine to perform the service requested by the CALL statement, a subroutine, which was attached to the supervisor or processor (element) at assembly time, is first entered. (Each Executive service has its own subroutine.) This subroutine simply places a certain code in a Store-and-Trap (STR) instruction, and then executes the instruction. The executing of the STR causes an interrupt (trap) in the 7094. The Executive fields the trap, interprets the code, and transfers control to the specialized routine designated by the code.

Multiprogramming aspects

As noted in the Introduction, the Executive is a multiprogramming system; i.e., it permits many independent paths of logic to proceed asynchronously and is able to switch control of the CPU (Central Processing Unit) from one path to another, depending on the priority of a supervisor or processor and its availability for a particular path. The priority of the supervisors and processors is determined by the order of its entry in the Executive priority table. This entry not only establishes the element's priority but reflects the general status of the element at all times by giving the following indications:

- Is currently operating or idle.

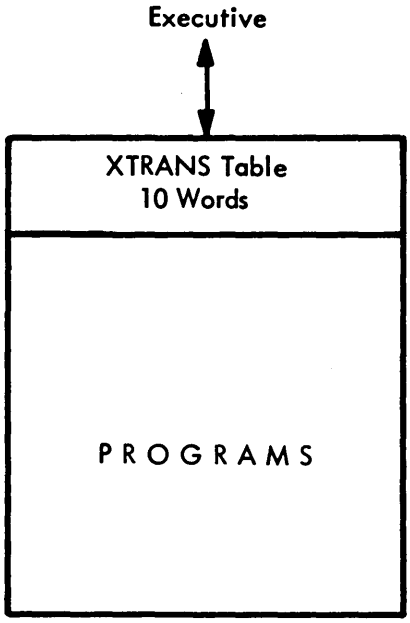


Figure 4.1 Standard Argument Area: XTRANS

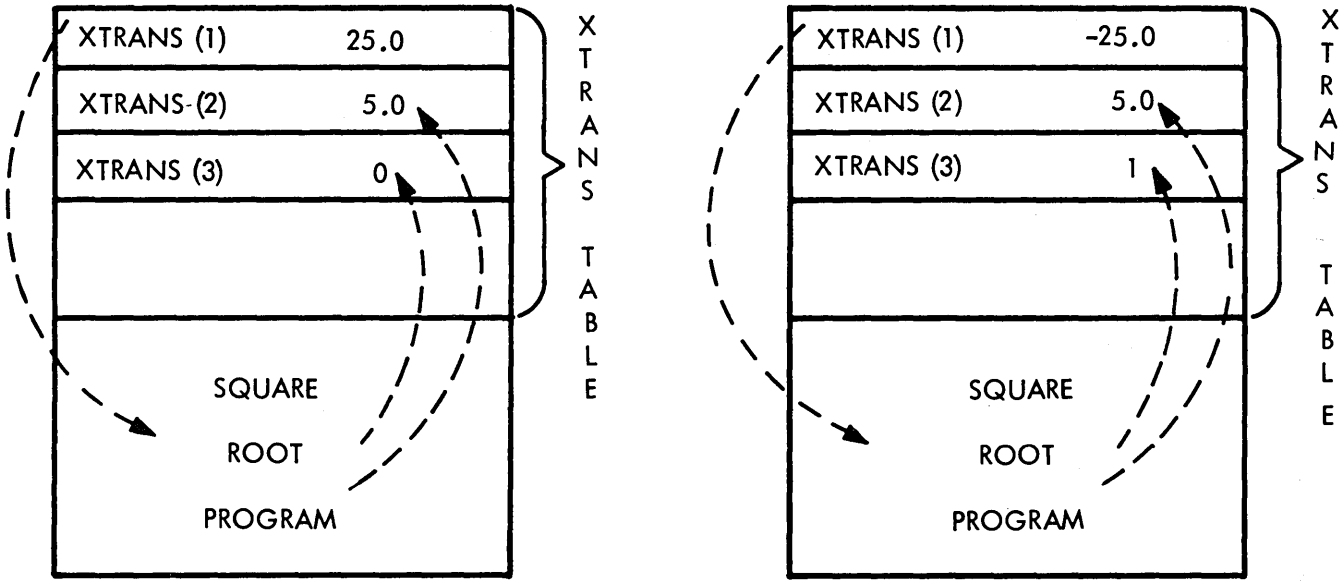


Figure 4.1 – Standard argument areas: XTRANS

Figure 4.2-4.3 – Square root program

- Has one or more XTRANS waiting in a queue to be sent to another supervisor or processor.
- Is being loaded into core.
- Is in core at location XXXXX, or is not in core.
- Is on LCS (program must be loaded from tape to LCS to core for execution; programs generally loaded into LCS from tape once per many core loads).
- Is a supervisor or processor.
- Is privileged (runs with Executive ignoring protect interrupts).
- Is suppressed from running.

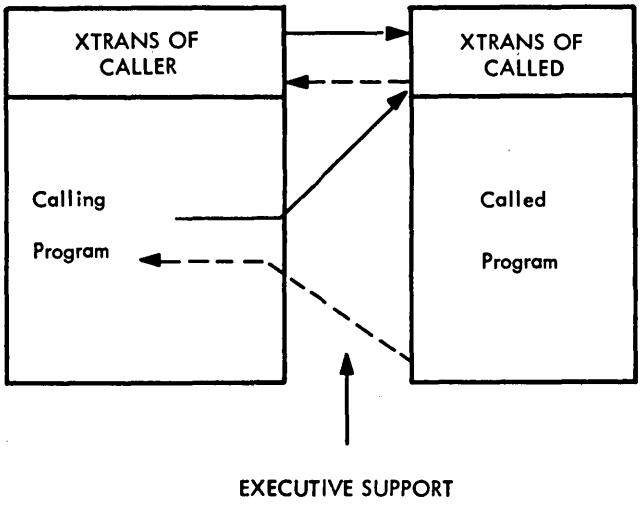


Figure 5 – Executive and XTRANS

The order of the entries in the priority table is established by the applications programmers through a macro at nucleus assembly time. During execution, Executive scans the priority table from the top each time a status change occurs. When Executive finds a program ready for execution, the scan stops, and that program is given control.

A supervisor further inhances multiprogramming in that it consists of one or more programming elements, called functions. Each function has its own XTRANS area and may operate independently; in addition, all functions share a single copy of a permanent data area kept for each supervisor in a special buffer called XTPERM. (See Figure 6.1.) XTPERM is permanent since the Executive preserves the contents of the table when the main core storage occupied by a supervisor must be made available for other uses. When the supervisor again receives control, the supervisor's XTPERM is exactly as the supervisor last left it. Processors have no XTPERM but many have temporary work space while executing. The size of XTPERM is established by the programmer to fit his needs for permanent data. There are probably no two supervisor XTPERM's the same size in the RTCC systems.

The second level of the two-level multiprogramming structure discussed in the Introduction of the paper is found in the functions of supervisors. Functions have an internal priority that determines which function is to receive control when two or more functions of a supervisor compete for control.

The function's design is based on the concept that a small package of functions (a supervisor) could effectively generate a number of parallel logic paths, and that multiprogramming will occur almost without

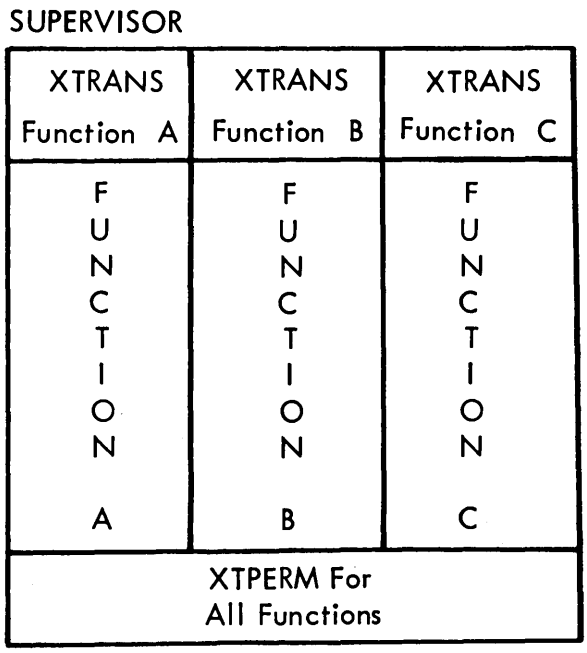


Figure 6.1 – Three-function supervisor

the programmer being aware of it. With a number of more-or-less independent logic paths operating asynchronously, the Executive can maximize the effective utilization of the CPU.

The supervisor function can call a processor several different ways. The classical method is to call a processor as a subroutine (see Figure 6.2). When the processor completes its task, it returns control to the function at the next instruction after the call. The function is out of operation, so to speak, until the processor completes.

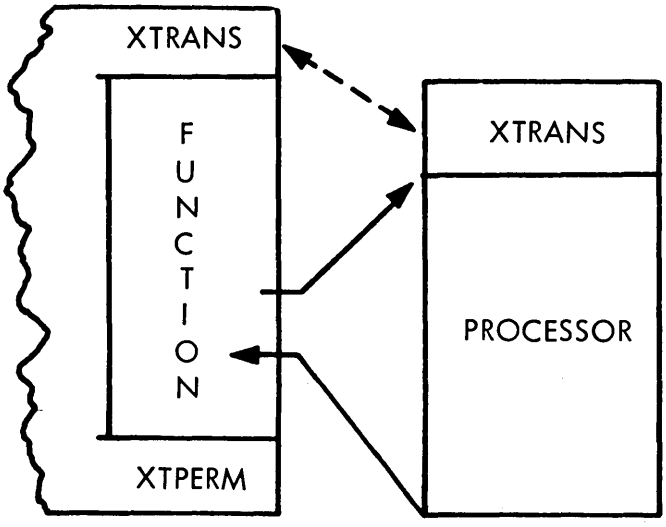


Figure 6.2 – Function of supervisor calling processor

Anotner approach permits the supervisor function to "send" a call to the processor so that the function does not give up control. Consequently, for calls that are sent, a function may call a number of processor (see Figure 6.3). In this method, a function may initiate a number of parallel operations. Furthermore, sending calls provides another control option. The supervisor function, in sending a call, can permit the processor to determine whether a return is to be made or not.

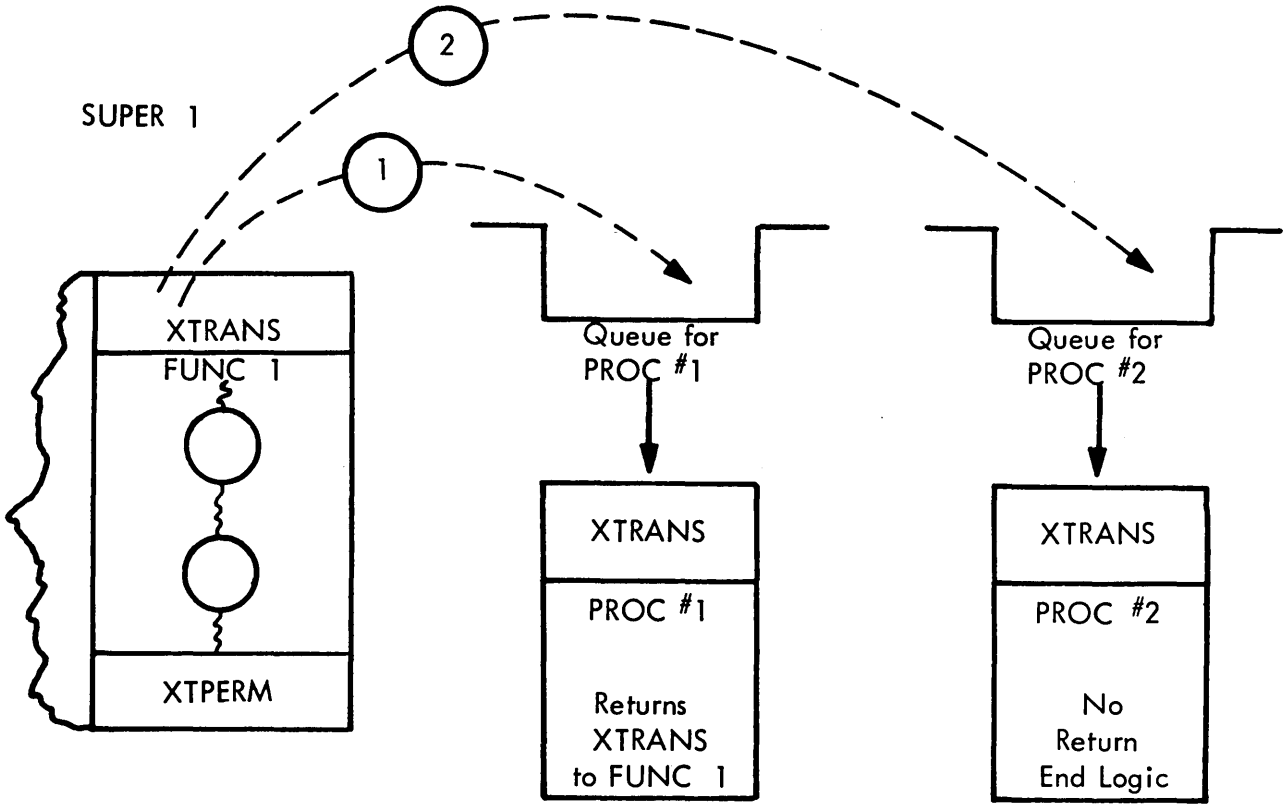


Figure 6.3 – Calls sent to processors

If no return is to be made, the processor represents an "orphan" task in multiprogramming that simply executes and completes the task entirely. This is a fairly typical operation where processors update D/TV displays and no return to a supervisor function is expected, that is, unless something unusual is uncovered in the processor's execution. If the processor returns, it must return to the start of the function specified by the arguments in a calling sequence of the original calling function. The function returned to may even be a function of a different supervisor. Therefore, the processor is effecting a transfer of control without knowing where this control is going.

Finally, functions of the same or different supervisors communicate by calls that transfer the XTRANS of the calling functions into the queue for the called function. If the calling function has the higher priority, control remains with that calling function.

Real time processing

Basically, we have placed the Executive in lower 65K core and stated that it performs allocation of supervisors and processors into main core storage from the LCS (more on allocation later in the paper) when a requirement for the supervisor and processor is known.

We have shown the supervisor (with its functions) and the processor giving request to the Executive for certain services. Now we turn to the major requirement for a supervisor or processor to be brought into operation. That requirement is the receipt of real time data.

Real time data receipt

In Figure 7 we find a processor in operation when a data channel trap (or interrupt) is received from the 7281 DCC (channel F). What has happened is that

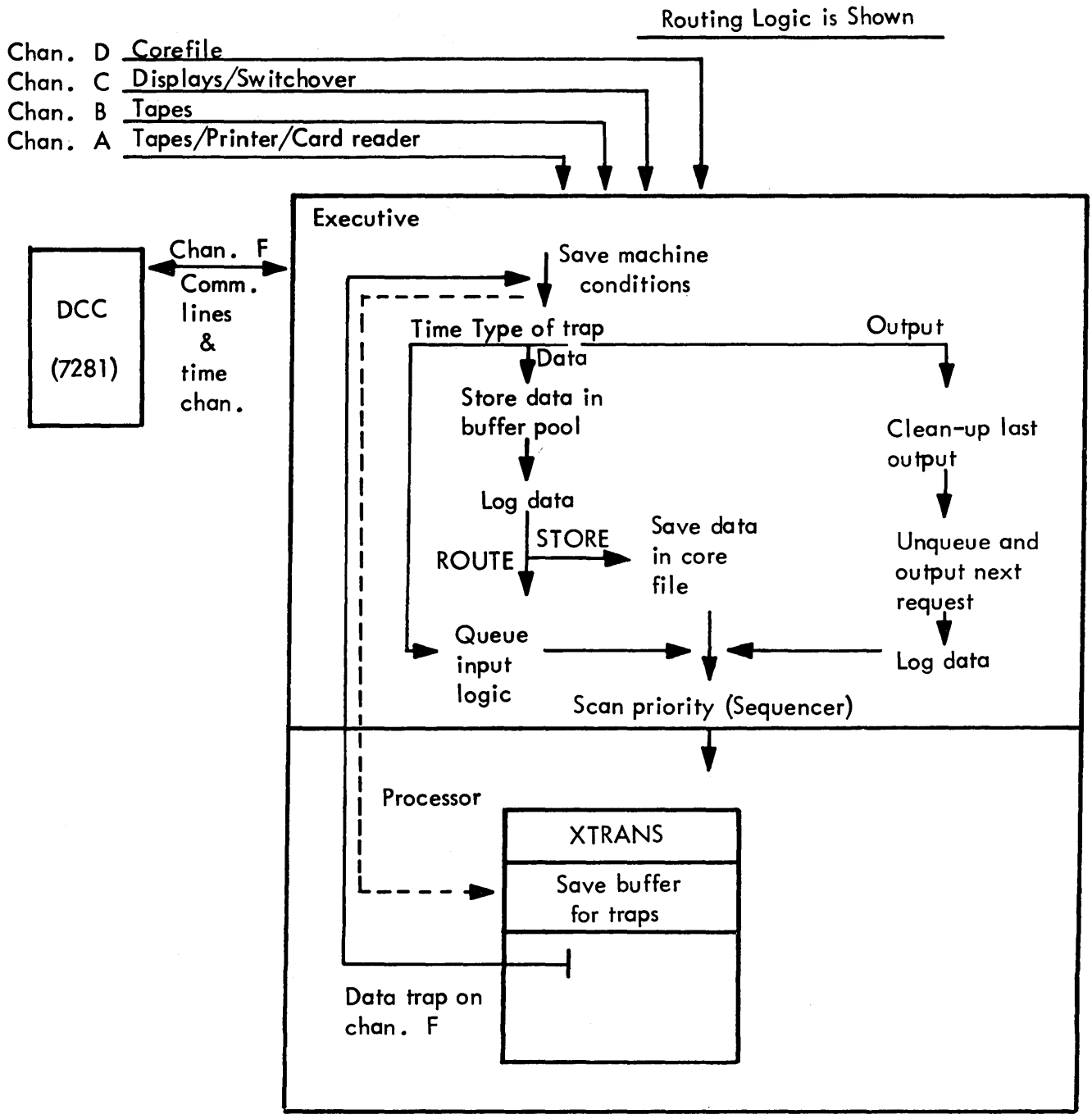


Figure 7—Trap control logic

data is now being placed in a buffer in lower core by the hardware. (Each of the DCC input subchannels has an addressed buffer in lower core.) The Executive receives control from the processor when the interrupt occurs and saves those registers and addresses that will permit return to the processor without change to the conditions existing before the interrupt occurred. Executive takes the data that has been placed in the input subchannel buffer and places it in a main core buffer pool for it to be logged onto an output tape. The other process that Executive performs on the data is termed routing and is covered in the next section. Now that Executive has received control via the data channel interrupt, it has an opportunity to scan the priority table to find the highest priority element (supervisor or processor) with a work queue waiting for it. The element found in the scan is then entered.

There are, of course, other Data Channel Interrupts, as shown in Figure 7, for channels A, B, C, and D.

Routing

The RTCC computers must interface with other computer installations and man/machine devices. Additionally, the RTCC computers must keep in step with Greenwich Mean Time (GMT) so results will be maintained in real time and will be synchronized with computing efforts elsewhere. The Executive routing feature manages the input from the communications lines (DCC) and routes data and time (GMT) to the proper functions and processors.

The routing logic is part of Executive; however, Executive makes no original decisions as to the destination of input. Routing information (directives) for time and data is supplied to Executive by the application programs. This information is stored in routing tables associated with DCC input subchannels. The user must activate and deactivate the directives by chaining.

When data arrives in the system, the data identification (ID) is compared to all possible ID's of data that might arrive over that subchannel. When the data ID matches an ID stored in a chained routing table, the routing table information is used to queue the data to an input function or processor, to store the data until a future time, or to discard the data because they are not needed.

When time arrives in the system, the current time (GMT) is compared to all routing tables which are used to direct queues depending on time. Routing will generate a queue to all functions and processors for which the request for time has been met.

Time signal routing

One type of routing is time routing. For instance, suppose a supervisor has to produce display output every second. The supervisor would inform the Executive that the supervisor requires control every second. The Executive would file this request in a routing directive for future reference. (See Figure 8.)

Every second thereafter, the Executive would notice, while scanning its time routing directives, the name of this supervisor listed as requiring a call every second. The Executive would create an XTRANS for the supervisor and would, in effect, call its type; i.e., time, and the current time. The type code would distinguish this particular XTRANS from any other types of XTRANS the supervisor may receive. (It should be noted that supervisors may call upon this supervisor with other type codes in XTRANS.)

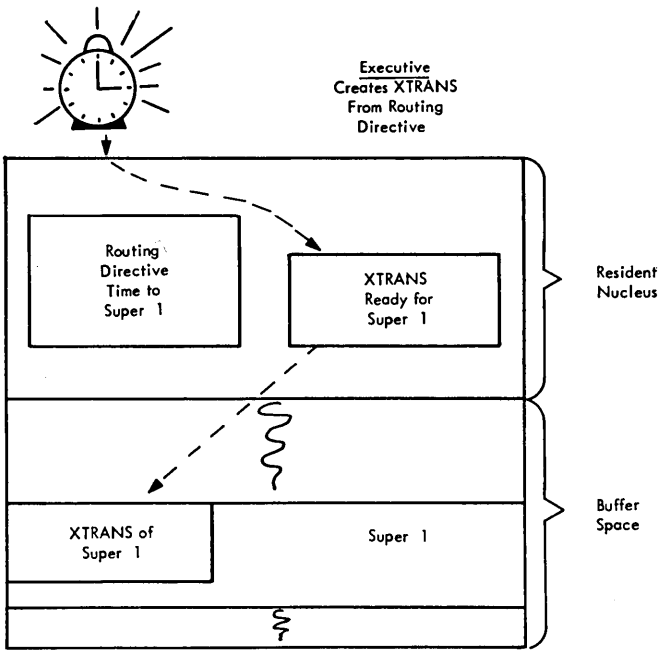


Figure 8 – Routing of time

It is particularly important to note that the supervisor need tell Executive only once that repetitive calls are required each second. The Executive will generate these calls every second, indefinitely, until the routing directive is modified or cancelled.

After the XTRANS is created, the Executive would attempt to give this XTRANS to the supervisor so the supervisor can begin processing. Frequently, a supervisor or processor cannot immediately receive the latest XTRANS because:

a. The supervisor or processor is not in main core.
b. The supervisor or processor is busy doing something else.
c. More important work, i.e., some other supervisor or processor has to be done first.

These problems are avoided, or at least deferred, by inserting the XTRANS into a queue for the supervisor involved. Every XTRANS, given to or created by Executive, enters a queue for the program that is to process the request. The queue is ordered chronologically; the earliest request is always at the top of the queue. When the program involved is available in core and has the highest priority, the request leaves the queue immediately. Otherwise, requests wait in the queue for their turn. Each time a program completes processing of one request, the program is available for the next request in the queue (see Figure 9).

Real time data routing

The Real Time routing in Executive brings real time data to the application programs. The data routing

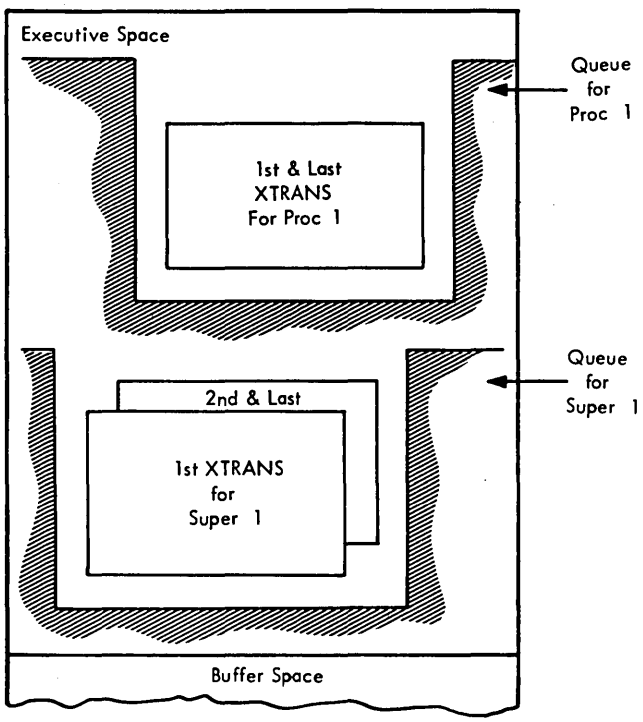


Figure 9—Queues of XTRANS

process is similar to the routing of timing signals. The application program specifies a routing directive for data. This routing directive is composed of two parts: the first gives Executive the criteria to identify the particular data the program requires, and the second part of the routing directive tells Executive what to do with the data that satisfy these criteria.

There are basically two options for applications programmers in routed data; direct routing, and store-mode routine. (See Figures 10 and 11.)

Direct mode routing

RTCC has few variable length messages. Messages of a given type generally have a constant size. Some types of real time data messages are small enough to fit within an XTRANS. For these messages, the programmer can specify direct mode routing. When a data channel interrupt occurs, Executive simply creates an XTRANS table, places the data into the XTRANS, and places the XTRANS into the queue for the program named in the routing directive. When the program receives the XTRANS, the data are then ready for processing.

If the data are too large for the XTRANS, the Executive places the data into a buffer in lower core and places information in the XTRANS that describes the location of the data in the buffer. Using the information provided in the XTRANS, the program obtains

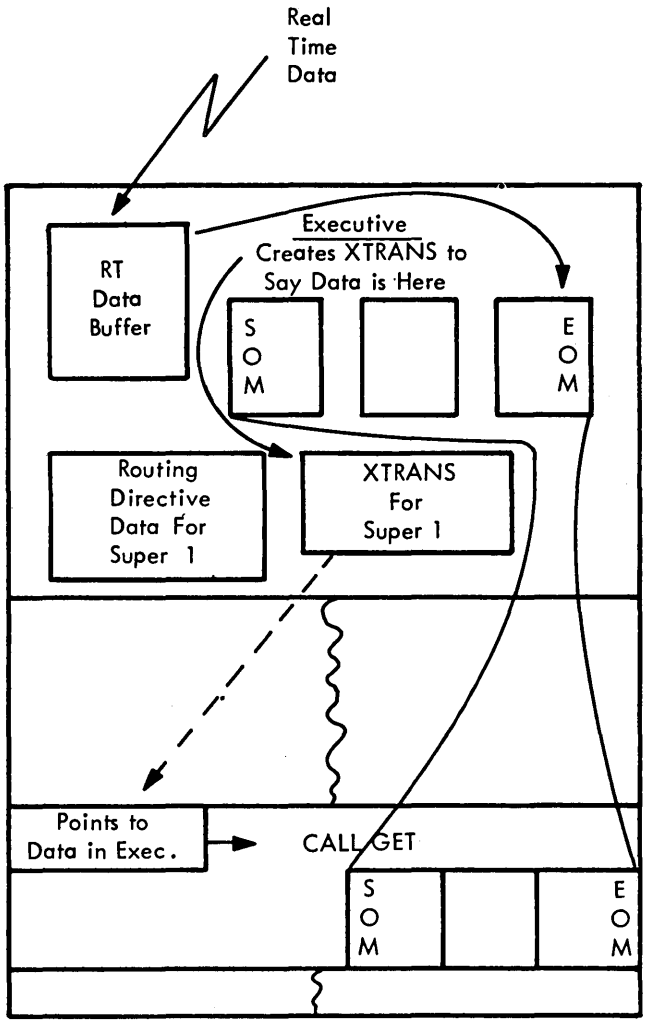


Figure 10—Direct mode data routing

the message by executing a CALL statement to request the Executive Real Time Input/Output Control System (RTIOCS) to move the message into the program's area.

The direct mode of routing (as demonstrated in Figure 10) is generally most effective for the small, non-repetitive real time data inputs.

Store mode routing

For cyclic real time data applications, processing usually consists of two phases: data collection, and data processing (see Figure 11). The data collection process can be processed entirely by the Executive. The programmer defines a routing directive that instructs the Executive to store the selected data into a data table on the LCS (data tables are termed Z-tables). The programmer also creates, separately, a routing directive that causes Executive to generate a periodic XTRANS as a function of time. When the

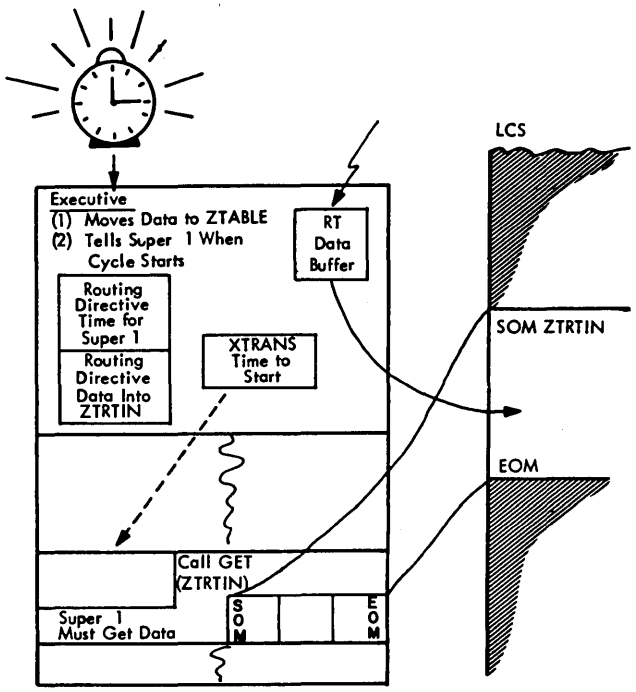


Figure 11 – Store mode routing

program receives this XTRANS, it requests RTIOCS to read the collected real time data from the Z-table into a designated area of the program for processing.

The major elements of executive

We have placed Executive in the Real Time system and shown to a limited extent how it provides the support and control necessary to sequence the execution of the various supervisors, functions, and processors through their request and through the receipt of real time data. We have spoken of Executive in general terms and definitions; now, we can turn to a brief description of some of the major elements that give Executive its structure.

Executive linkage

Executive linkage with supervisor/processors is effected through library routines appended to supervisors/processors at assembly time as the result of CALL statements within the supervisors/processors and associated service routines within Executive. As has been shown previously, when an Executive capability is called by a supervisor/processor, the appropriate library routine in the supervisor/processor sets up and executes an STR instruction that contains a numerical code identifying the type of call. Execution of the STR instruction causes control to be passed to Executive under hardware control. Executive references the code in the STR instruction to determine the type of call and then passes control to the associated Executive service routine.

Executive sequencer

The Executive Sequencer consists of those routines within Executive which service the real time system by:

a. Interpreting Central Processing Unit (CPU) traps which occur from the execution of an STR instruction within a relocated element.
b. Servicing floating point and protect CPU traps.
c. Assigning control to the highest priority element which has work outstanding.
d. Saving and restoring machine conditions when a data channel trap or CPU trap occurs.
e. Interpreting requests for transferring control between the elements of the system.

Real time input/output control system

The purpose of the Real Time Input/Output Control System (RTIOCS) within the Executive is to provide a simple (from the programmer's standpoint), flexible communication link between the supervisors and processors and the various input/output media available in the RTCC (no "Raw I/O" is allowed outside the RTIOCS). The RTIOCS consist of a series of integrated routines which perform all necessary input and output functions to the following devices and storage media employed within the RTCC: Tapes (Channels A and B), 7286-II 512K core file (Channel D), 7281-II Data Communications Channel (DCC, Channel F) which consist of 13 subchannels connected to such devices as plotters on the output side and real time data receivers on the input side, the Digital/TV System (Channel C), 65K primary main core memory, printer and card reader.

The basic framework of the RTIOCS, from the user's standpoint, is a statement CALL I*GETT or I*PUTT (* = "R" if the call is made by a processor and "U" if made by a supervisor) and a series of three to five arguments showing the action to be taken. A typical call might be: CALL IRGETT (ZXAMPL, MYBUFR, NBRWDS, LOCINB, BLKNUM). This translates to: from data file ZXAMPL, starting at LOCINB (a symbol containing an integer) in BLKNUM (a symbol specifying a block number if this is batch data) transfer NBRWDS (a symbol which contains the number of words) into MYBUFR (a symbol for a buffer area, normally within the calling processor). The data file name termed Z-table name (in this example, ZXAMPL) is a symbolic name of a four-word table (File Control Block) in Executive which defines the data file (its location, its type, its

size, and other information pertinent to the particular device that is the data file). Data files may be on tape, LCS, main core, etc.

Figure 13 gives an example of RTIOCS servicing a request for I/O from a user. The same type CALL service logic using the STR instruction that was discussed earlier in this paper is used for I/O request.

DCC servicer

The DCC Servicer processes all 7281-II DCC subchannel traps. These traps may be caused by both the input and output subchannels. The DCC Servicer moves the data from the cells in lower core in which the hardware placed the input data into the Executive Buffer Pool, sets up information for the Executive logging routines to log the data, and sends a request to Routing to route the data.

Dynamic main memory and auxiliary storage allocation

It was expected that the Gemini systems would change from mission to mission and would grow to exceed the capacity of the computer main memory. Since this change and growth could not be contained, the necessary flexibility was built into Executive to permit such change and growth. Part of the flexibility is in the design of the storage allocation routines of Executive. No permanent storage location is assigned to any problem program. Storage is allocated on demand and in the quantity necessary to accommodate the particular program.

Two distinct levels of storage allocation are used. The first, allocation of main memory, is essential to every run. It provides for the allocation of areas of memory to required relocatable supervisors and processors and uses the Executive RTIOCS capability to load the programs into memory from the LCS (see Figure 12).

The second, LCS allocation from magnetic tape, is necessary only when there are more relocatable programs than can be accommodated concurrently in the LCS. It provides for the allocation of areas in the LCS to relocatable programs (processors only) according to both actual and user-anticipated requirements. It also supervises the transmission of the programs from magnetic tape to the LCS.

When a request for a supervisor or processor is made, the Executive Sequencer determines if it is in main memory. If it is not, Sequencer makes an explicit request for the program by transferring to the Executive allocation routines. If the program is available in the LCS, the main memory allocation routine attempts to allocate memory for it. If the program is not in the LCS, the LCS allocation program is queued to bring the needed program from magnetic tape to the LCS, after which memory allocation will be re-attempted.

Once main memory has been allocated for a program, the Real Time Input/Output Control System (RTIOCS) reads the program from LCS into main memory. Control is then returned to Sequencer. When no main memory can be allocated for a program due to relative priority, activity status, and length considerations, the request for allocation is retained so that it may be re-attempted later.

LCS allocation is initiated in response to actual requirements for programs to execute in main memory or in response to user calls that specify which programs will be placed and held in the LCS in anticipation of actual requirements.

Initializing the real time system

The final element in a discussion of the Executive is Initialization. Executive initialization provides user options for the Executive nucleus to execute in several modes and in various hardware configurations. Initialization occurs prior to entering Executive and prior to starting a real time or simulated real time operation. The initialization options permit distinctions between real time or simulated time, real input data or simulated data, and actual I/O devices or simulated replacements. The general scope of initialization includes:

- Establishing initial hardware conditions
- Establishing parameters and initial conditions for storage allocation
- Ensuring proper linkages between certain real time programs
- Assigning tape drives
- Loading initial data into data files on the LCS (Z-tables)
- Creating the Executive buffer pool
- Establishing debug request tables.

Prior to entering into the discussion of Initialization, it is essential that we give the two steps the user must take before his supervisor or processor is in an application system that is being initialized. The first step is to accomplish unit testing of his supervisor/processor and the second is to create the application system tape.

Job shop simulator

Since all the Executive services are provided via the CALL statement, a simulator of the real time environment was easily provided under the IBM 7094 IBJOB system. This capability permits testing of a

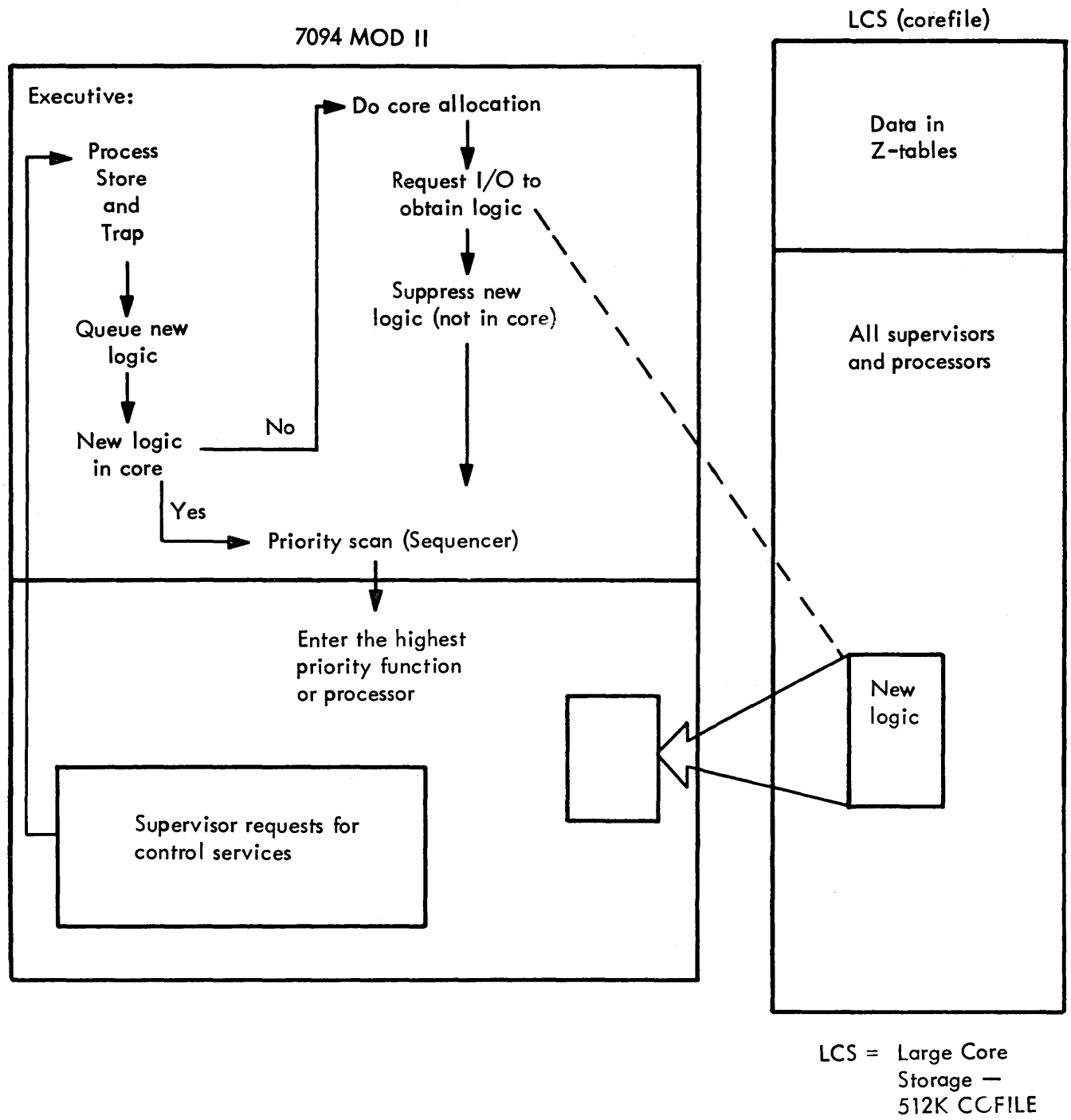


Figure 12—Program control flow

single processor or supervisor, or a supervisor and several processors, in a batched-job system. Some of the more exotic features of Executive's multiprogramming facility cannot be simulated adequately in a sequential (essentially IBJOB) environment. But for most unit, string, or subsystem testing, the Job Shop Simulator is very effective. The fact that processing is sequential often makes it possible to identify bugs before the environment changes completely (a constant problem in multiprogramming debugging). In addition, the capability of conducting significant debugging in a batch environment greatly economizes the computer time required to deliver checked out systems.

Creating the real time systems tape

When unit testing has been completed with the Job Shop Simulator, the programs can move unchanged

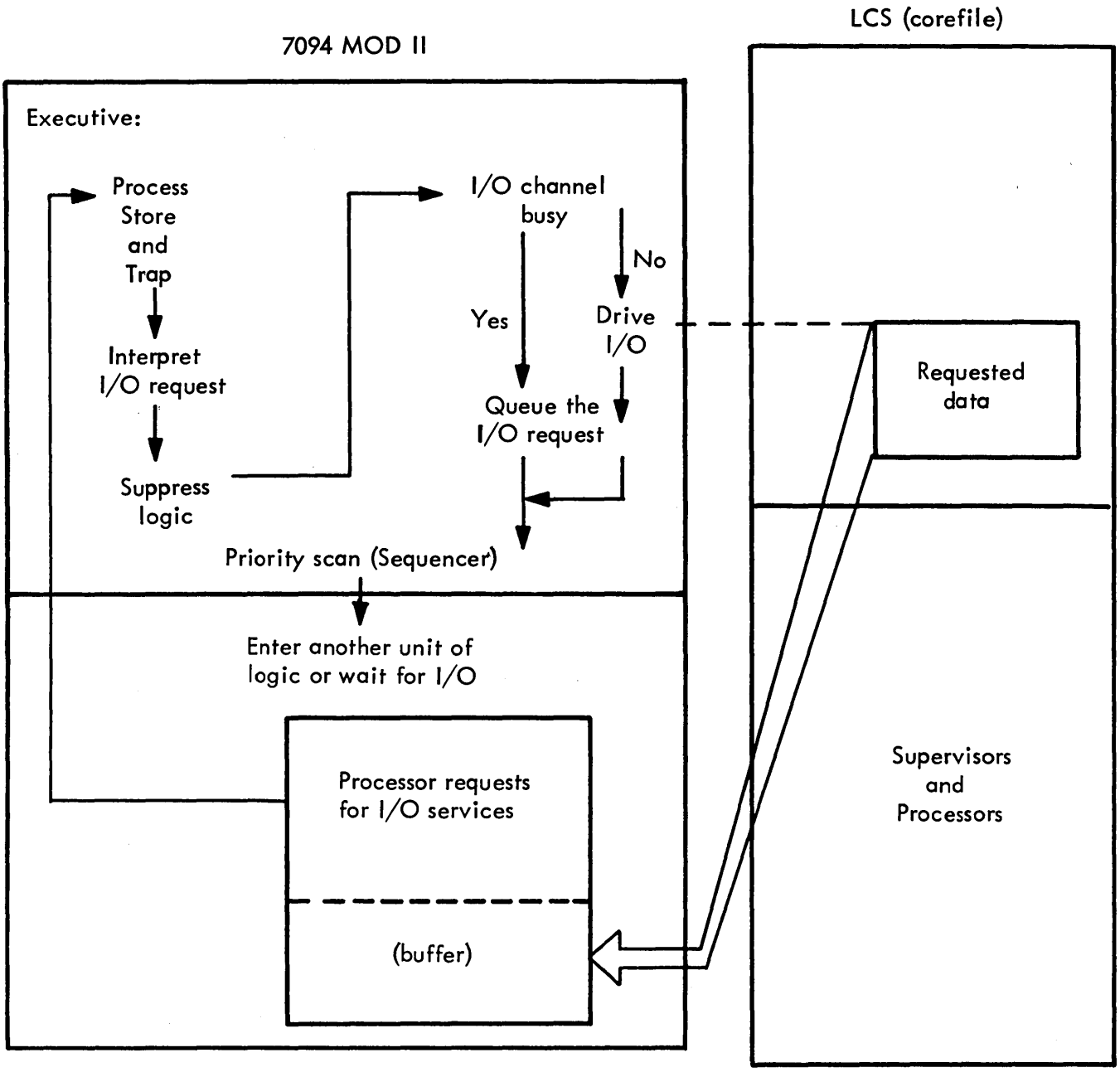


Figure 13 – Real time IOCS

into the real time system testing environment. The first step is to create a real time system tape.

The Executive has been described as a single-computer program serving many real time applications systems. The user must define his application system to the Executive by building tables inside the Executive. This is accomplished by macro statements that the user inserts into a special Executive card deck. When this deck is assembled, the user's section of Executive is created. These macros basically define:

a. The supervisor/processors of the application system and their relative priority (the priority table)
b. The file control blocks (for Z-tables)
c. The initial routing directives.

During the building of the system tape, the user's decks are combined with the Executive code to produce the Executive nucleus. In the process, all of the symbolic names (program names, Z-table names, etc.) are translated into indexes. The translation

process simply trades pre-execution time to save translation in real time. The nucleus Executive is written on the real time system tape along with copies of the remainder of the non-resident Executive and all of the user's application system programs.

Real or simulated time

The first significant Initialization option is whether or not the user requires an internal/external clock synchronization. Unless external devices (including other computers and/or people) are involved, synchronized time is rarely used. Unsynchronized or simulated time simply uses an internal clock that never runs when the computer is idle. When idle time occurs, this clock is spaced forward to the next clock interrupt. For example, this feature permits an orbit of 90 minutes to be completed in about 10 minutes of elapsed time. No changes are required in any applications system program. In fact, there is no way an application program can tell that a simulated clock is being used.

When running in the simulated mode, the user can specify that any or all of the real time input devices (subchannels) are to be simulated with canned data from tape or, in some cases, the card reader. Any of the real time input devices can be simulated while the remainder accept actual data, or are unused, in any combination. Furthermore, the real time output devices can be used or the outputs can be diverted to the LCS by requesting on-line for Initialization to modify the file control block.

Debug request

The RTCC version of the IBJOB debugging system that permits core snapshot dumps, heavily used in job shop runs, is also available when running a real time system in the simulation mode. Since the debugging package operates with the simulated clock turned off, the applications programs cannot recognize that the debugging operations are taking place.

When a real time run is specified, initialization automatically removes any debug requests.

Error halt or error recovery mode

The normal inclination of the real time Executive is to continue processing, regardless of any errors that may occur. Some error recovery action is instituted in hope that the condition was only transitory. This is the sensible approach to real time support. But in debugging a system, especially a highly dynamic multiprogramming system, evidence should be saved as soon as the error is discovered. Consequently, another initialization option permits the system to run under either the error halt or the error recovery mode. Once this system is started in the error halt mode, the mode may be changed to error recovery and back, at the setting of a sense switch. Once the system is started in the error recovery mode, the mode may not be reset.

Real time statistics

Another initialization option permits the user to accumulate statistics during a real time run. The accumulation of statistics operates only in the synchronized real time mode and generally requires about five percent of the CPU in overhead. (The RTCC experience has been that the five percent of the CPU is not the difference between success and failure in a real time run.)

Once the initialization option has been set, the Statistics Gathering System (SGS) may be activated or deactivated dynamically from the Manual Entry Device (MED) or by the card reader used to simulate the MEDs. Statistics are accumulated in three categories:

a. Internal Executive Logic—frequency of use, average execution time, core allocation attempts and successes, etc.
b. Supervisor and Processor—number of uses, average execution time, number of uses per time loaded into core from the LCS, number of Executive CALL's, etc.
c. Total CPU Utilization—amount of time in execution, in waiting on I/O, and idle.

SGS, originally conceived and implemented to support the extensive GPSS (Gordon General Purpose System Simulator) modeling activities at RTCC, has proved useful to many of the applications programmers in analysis of their systems.

Operational features of executive

Real time run synopsis

When the run terminates, the user has the option of:

- A *synopsis* that consists of a formatted presentation of all the significant Executive tables: the state of the priority table, the state of all the processors and supervisors, the chains of XTRANS in the queues for all the processors and supervisors, etc.
- An octal dump with assembly language operational codes.
- A full symbolic dump.

If any debugging system snapshots were taken, these are formatted in the post-execution processing. Programmers generally take the synopsis and an octal dump.

Real time internal control mode and generalized on-line display capability

The Real Time Internal Control Mode (RTICM) of Executive gives the user of Executive more control of the multitude of initialization and dynamic options, while giving the user the capability to exercise these options remotely. Most of the Executive options, prior to RTICM, required the user to set and reset the sense switches and keys of the 7094 console. RTICM permits all options to be punched into cards and allows selective dynamic use of these cards to be governed by the processing itself. The on-line display capability allows a user to select dynamically, via the Manual Entry Devices, information from main core on the LCS to be displayed on the television system. The information can be selected symbolically, saved, and recalled by name.

Reliability

During the Gemini and Apollo missions, the Executive is keyed to keeping the real time system up and running no matter what adverse conditions are encountered. In doing this, the Executive has provided an elaborate Error Recovery System to intercept program errors, hardware generated errors, and data generated errors. When one of these errors are encountered, Executive quickly examines the situation and produces an appropriate recovery method to enable the real time processing to continue. If the error encountered is of such a nature that recovery is either impractical or unfeasible, the Executive will recommend Switchover to a standby system. If necessary, the 65,000 words of core memory and the 524,000 words of COFIL memory can be transferred from this standby computer to a new operational computer by the Executive RESTART logic in less than five minutes. In the worst case, real time processing is never delayed more than three minutes. If an I/O device fails in any manner, Executive provides time-out logic to ensure that failure on one device will not interfere with the remainder of processing in the real time system. If tapes fail or become full, Executive provides tape switching logic.

CONCLUSION

Some of the basic ideas for the Executive were developed in the Real Time Mercury Monitor for NASA's Project Mercury and, in turn, some of the ideas conceived in the Executive design are being used in the development of the Real Time Operating System/360 for Project Apollo. It has been found in these endeavors that real time system development is an evolving creature, for the predominant requirement in its development is that its design must be able to evolve as the environment in which it will operate is understood.[2]

REFERENCES

1 J H MUELLER
The philosophy of the RTCC control programs
IBM Real Time Systems Seminar Proceedings 1966

2 R L HOFFMAN
Managing the design, development, and implementation of large scale generalized real time systems
IBM Real Time Systems Seminar Proceedings 1966

Executive programs for the LACONIQ time-shared retrieval monitor

by *D. B. J. BRIDGES*
Lockheed Palo Alto Research Laboratory
Palo Alto, California

INTRODUCTION

LACONIQ* was designed to give several users who are not necessarily familiar with programming the apparently exclusive on-line use of a small computer for processing large information files. Processing of the data may include manipulations such as retrieval, updating, or the deletion or creation of records, documents, files, etc. Thus, LACONIQ is general-purpose in the sense that many types of information files may be processed, but is special-purpose in the sense that scientific computations on-line are not necessarily intended to be practical.

Considerations which led to the major design decisions for LACONIQ are given in reference 1. The introduction and bibliography from an early version of reference 1 are reproduced by Appendix A. Very briefly summarized, the more important features of the system are its multilevel structure (user-application-monitor), the event-driven rather than clock-driven nature of the system, the polling of consoles at the convenience of the system, the avoidance of program roll-out to a peripheral device, and the scheduling of resources for a small computer.

This paper is primarily concerned with the development of LACONIQ executive programs on an IBM 360/30 computer. The four sections immediately following are principally concerned with overall system considerations including hardware, software, and pertinent system events. The remaining major sections proceed from the general to the detailed regarding a central executive scheduling program for LACONIQ.

Hardware

LACONIQ is programmable on any small computer that has moderate capabilities for programmed and I/O interrupts, storage protection, base registers, etc., such as the IBM 360.[2] An IBM 360/30 with 32k bytes of 1.5 μsec core was chosen for experimental exploration of the LACONIQ design. The hardware configuration is shown in Figure 1.

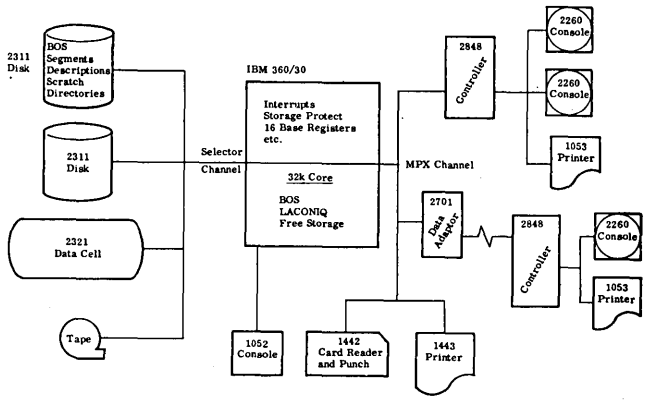


Figure 1—Hardware for initial LACONIQ implementation

In the LACONIQ system, core is divided into three main areas. One area contains the IBM Basic Operating System (BOS) and a second area contains the LACONIQ system programs. The remaining free storage is dynamically allocated to application programs, scratch areas, and internal buffers.

The minimum peripheral storage needed for LACONIQ is one disk. The disk must contain BOS and areas assigned for application programs. The disk may also contain areas for working storage, file directories, and data. Ordinarily, large data files would be stored on a bulk storage device such as an IBM 2321 data cell. Two disks and one data cell were used on one selector channel for the initial LACONIQ implementation because information retrieval from a large file was one of the prototype system applications.

The on-line consoles used for LACONIQ contain recirculating buffers large enough for one full CRT

*LAboratory Computer ON-line InQuiry monitor.

screen display of 960 characters. The presence of this buffer allows LACONIQ to interrogate each console at a convenient time from the point of view of the system. The initial hardware configuration included three IBM 2260 CRT consoles of which one or two were attachable at a remote location.

A small printer was attached at each controller principally for user-initiated display printout. The CPU console, line printer, and card reader attached to the multiplexer channel are used by BOS and some of the LACONIQ diagnostic procedures.

Types of LACONIQ programs

Since the system is oriented toward a particular series of applications (information processing), all programs might be considered as application programs. However, a division of the LACONIQ software into two major levels, application and system, enables the use of the system to be more conveniently specialized for each of the several different types of processing which may be desired on the several different information files accessible to the computer.

The first level of software, with which the unsophisticated console user interacts most directly, consists of prewritten application programs especially tailored to efficiently process data from his file or request input (output) from (to) his file. The second or inner-most level of software consists of system programs designed to satisfy the requirements of all users on a resource-shared basis.

The system programs and application programs have some conventional characteristics. In general, a system program is general purpose, frequently used, and resides permanently in core. An application program is special purpose, less frequently used, and resides on disk storage.

A third class of programs in LACONIQ is called application-zero. The name is derived from the assignment of I.D. numbers to LACONIQ applications in the series 1,2,3 .... Zero is used as the I.D. number of the system-type application programs mainly for ease in verbal reference. Application-zero programs are general purpose (system) in function, but reside on disk to save core space at negligible degradation of response time (because they are relatively infrequently used).

Application programs

Application programs are written in assembly language except for I/O.* There have been three principal pilot applications written to date. One of these concerns text writing, updating, and retrieval from a data bank of engineering documents. A second is concerned with processing of failure data on missile component parts. Another application finds facts from a file of information concerning military capabilities and resources. A set of disk-resident programs is written especially for each data file and particular processing goal. LACONIQ imposes few restrictions on data file organization. Application programs include the capability for on-line dialogues of the type described in reference 4. Generally, these dialogues give the user the result of each significant processing step and provide options for continuing the process or for returning to a previous step. The user's response is frequently a single key stroke or the typing in of a word, but in textwriting applications the user could type in several hundred characters. Typically, application programs include optional tutorial dialogues to acquaint the console user with how to use the system to manipulate his data file.

The applications are programmed in separate segments which are brought into core one at a time for each console. Each segment is executed to completion while residing in core. Its execution may be interrupted by its own requests for service or by I/O interrupts.

Also brought into core with each application segment is a self-descriptive section and the identification of all possible successor segments. These descriptions, written by the applications programmer, contain data such as segment length and block size. The descriptions are stored in a reserved area of core which includes a communication area (for transfer of data from one segment to the next) and an input area (for short console input strings).

System programs

The LACONIQ system programs and their principal interconnections are shown in Figure 2. The central scheduler performs various housekeeping tasks, makes entries in I/O queues, assigns priorities, and gives control of the CPU to system programs or application segments as required. The functions of the two programs to the left of the central scheduler (Figure 2) and the two programs to the right are parallel. The programs on the left resolve I/O needs for console communication and the programs on the right are concerned with I/O for peripheral storage. The latter include input of both data and program segments as well as data output.

The programs horizontally adjacent to the central scheduler perform the task of scheduling I/O requests for particular devices. If these devices are not busy, the schedulers call their respective control programs which issue the required CPU instructions and channel commands to initiate a request.

*The use of a higher level data manipulation language such as ALTEXT[3] is currently under investigation.

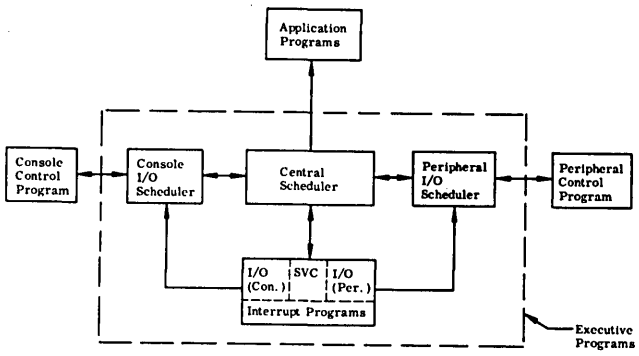


Figure 2—LACONIQ system program organization

A complete discussion of the scheduling algorithm for peripheral devices is given in another paper.[5] Very briefly summarized, it schedules I/O for slow devices before fast devices and gives preference to queued requests which require a minimum physical (e.g., disk arm) movement from the current position.

The scheduling of console output requests is handled partly by the central scheduler and partly by the console scheduler. When the console scheduler is called, it checks the controller line status and if the line is not busy, it passes the next request in the queue to the control program.

The two principal types of interrupt programs are for Supervisory Call (SVC) and I/O interrupts. The SVC on the IBM 360 series machines has several functions. One is to cause an interrupt with up to 256 different programmed interrupt codes possible. LACONIQ currently allows 16 of these codes to be used by application programs for service requests. Requests for I/O and other service needs are communicated to the executive routines via SVC interrupts. The I/O interrupt routines for consoles and for peripheral devices update various status indicators whenever channel or device end interrupts occur.

Not shown in Figure 2 is the program which dynamically allocates and releases free core storage. The storage algorithm has been described in another paper.[5] Briefly it involves look-ahead procedures utilizing the application program segment descriptions. This allows division of free storage into occupied and usable areas; the usable area is divided into available and storage-in-waiting.

Allocation requests originate principally in one or another of the three schedulers. Release of program, scratch, and peripheral input storage is usually set up by the SVC exit routine. The application segment requests the SVC exit routine when it has finished execution. Storage for console input is released at input interrupt time provided input is less than 33 characters (the usual case) when the input is stored in a console-dedicated area. Release of output buffer storage is usually requested by a routine which handles the interrupts when all output is completed.

Application-zero programs

Examples of disk-resident system programs are those which output the initial "hello" display to a console, analyze console input, and perform diagnostic procedures.

The successor segment to an application segment which has completed its execution may be uniquely identified in the appropriate successor description area, or its selection may depend on user console input. In the latter case, the input analysis program is called into core. The console input must either match specific strings contained in the description area or fall into a category of input described as legal for the completed segment. The category of input might be, for example, a string of alphabetic or numeric characters containing no arithmetic operators or special characters.

Diagnostic procedures in LACONIQ include various failure levels. For example, if an error is detected in the execution of an application segment, the user is usually given an opportunity to re-try. If the error persists, the user may be told via a display that his console is temporarily not operable.

Use of IBM basic operating system

The core-resident LACONIQ system programs all look like problem programs to BOS. They are assembled and linked following the ordinary rules for BOS modules. The BOS Linkage Editor resolves all relocations and the BOS Job Control program puts LACONIQ's initialization program into execution. The initialization program reads the hardware specifications (describing the current peripheral environment and on-line consoles) in from cards, modifies the new program status words (so that LACONIQ will capture the interrupts), and puts the processor in the supervisory mode so that LACONIQ programs can execute privileged instructions. The complete status of the System/360 is contained in a program status word (PSW). Each class of interruption (program, supervisor call, external, machine check, input-output) has two fixed locations in main storage: one to receive the old PSW when an interruption occurs, and the other to supply the new PSW that governs the servicing of that class of interruptions. In the IBM 360 certain instructions may only be executed by a supervisor program. Examples are instructions for I/O and instructions to set storage protection keys. If these instructions are executed in the nonsupervisory mode (problem mode) a pro-

gram check occurs and execution of the offending programs stops.

The major parts of BOS which are used on-line after LACONIQ initialization are physical IOCS and certain error-handling routines for peripheral devices. By maintaining its own status indicators, LACONIQ never causes the queuing procedures in BOS to be invoked.

System events

The two principal types of events in the system are the SVC and I/O interrupts. Requests for I/O and other service needs of a segment are communicated to the executive programs via SVC interrupts. It is considered illegal for a segment to issue I/O control instructions or to use physical IOCS.

The SVC interrupt routines interpret an interrupt code which reflects the type of service desired. LACONIQ provides SVC codes for the following kinds of service:

- I/O for peripheral devices*
- I/O for user consoles
- Allocation (release) of scratch core areas
- Movement of data
- Point-to-segment description areas
- Exit (and performance of housekeeping functions)

The SVC exit routine signals one of the fundamental events in LACONIQ: the segment has finished execution and the core areas it required may be released. A more common practice in time-sharing systems is to give each program (segment) a time partition of arbitrary length. (See Biblography and Appendix A.) When the segment executes for the specified length of time, it is interrupted and may need to be rolled out to disk (and later rolled in) if its core space is needed by another program. As explained in reference 1, in addition to avoiding roll-out, completion of each segment implies fast response time for simple tasks and somewhat longer response time for more complex tasks.

I/O interrupts are accepted whenever application segments are in execution and periodically when the central scheduler enables interrupts momentarily. The console or peripheral interrupt routine executed at interrupt time posts various status indicators for the executive programs.

An I/O interrupt of particular interest is the interrupt resulting from the depression of the ENTER key* by a user on a local console. This interrupt is posted and handled later at a time convenient to the system. Remote consoles do not generate this interrupt so they are periodically polled (whenever input is pending) to check if the user has depressed the ENTER key. One effect of the postponement of handling of local console interrupts is that all consoles, whether remote or local, are given the same kind of service.

Other events occur for external and program check interrupts. The external check usually signals that a segment has exceeded its maximum allotted execution time (a rare occurrence) and the program check invokes diagnostic procedures. The storage protect feature prevents an application segment from writing in the LACONIQ system area of core. The chance of a program error occurring on-line is considerably reduced by requiring that applications be checked out with an off-line program called LACSIM which simulates the LACONIQ environment.

A typical sequence of events in the system is given in Appendix B.

Executive schedulers (general considerations)

It is important to be efficient in use of the CPU in all on-line system programs, particularly in the operation of the central scheduler. Several dominant reasons are:

- The efficient use of the CPU is of interest for non-I/O-bound situations which may arise in a given situation, application, or configuration. For example, LACONIQ could be used with very fast peripheral devices, or bulk core, or in applications which are ordinarily compute-bound.
- Overhead costs to on-line console users are reduced.
- More CPU time is available for background processing.
- An endless variety of auxiliary tasks can be performed in CPU time not required for the more fundamental system functions.

Examples of auxiliary tasks are the gathering of system performance statistics and dynamic adjustment of system parameters to attempt to optimize system performance. Examples of tasks of the last type are updating of priorities for requests in queues, adjustments to ensure equitable distribution of resources to remotely located consoles whose service is degraded by slow transmission rates, policing of application segments in a partially checked-out status, and controls on maximum response time to console requests based on elapsed clock time.

The general LACONIQ approach to efficiency in CPU usage has been to maintain rather detailed status

*At the present time, the application programmer must write SVC requests for peripheral I/O for particular devices. Work is underway to allow SVC's for this function which are device independent.

*All CRT communication devices have one or more keys which the user depresses to indicate that he has finished his keyboard input. This key is called the 'ENTER' key for the IBM 2260.

indicators pertaining to the work-in-progress for each console. Various status indicators are up-dated throughout nearly all system programs to be interrogated by the central scheduler. The central scheduler gives control to a system program or an application segment depending on the configuration of the status indicators. An alternative to this procedure which was originally considered (but discarded because of predicted inefficiency) was to have the central scheduler give control of the CPU to various system routines on a periodic basis (rather than a known-need basis). Each system program would then search various queues and indicators to find something to do. A major disadvantage to this approach is the time wasted searching for a task when there is nothing pending.

In the current system, the peripheral scheduler (PIOS) is the only system program which can get control of the CPU when a need does not exist. Due to the peripheral I/O-bound nature of the retrieval process, PIOS always gets control following peripheral I/O interrupt handling so it can immediately schedule another request (if one is pending—the usual case).

System status information

The system core lay-out in Figure 3 serves as a summary of the principal status indicators in the system.

The selector channel status contains a busy bit for the channel and a busy bit for each device attached to the channel. In general, these indicators are set "on" by the peripheral control program when I/O is initiated, set "off" by an interrupt routine when I/O is completed, and interrogated by the peripheral scheduler.

The multiplexer channel status area contains one bit for each subchannel. This bit is interrogated by the console scheduler, set to "busy" by the console control program, and set "free" at interrupt time.

There is one peripheral I/O queue for each peripheral device. Each entry in a queue contains bits to indicate whether the request is in process or finished.

A 32-bit status word interrogated by the central scheduler (called WIPS for work-in-progress scheduler) is contained in the system part of each console-dedicated area.

The console status word

The significant status indicators which are grouped (for convenience) into a single 4-byte computer word are identified in the left-hand column of Table I. Each indicator is implemented either as a single computer bit (for YES-NO or ON-OFF type conditions) or as a group of bits (for the numerical count of the number of pending I/O requests) as shown in the second column from the left. The remaining columns of the table are headed by names of the principal executive programs which update the status word, i.e., turn an indicator on/off, or increment/decrement a count. A "1" entry in the table indicates the program at the top turns on the indicator identified on the left. A "0" entry indicates that the program turns the indicator off. An "up" arrow indicates that the program increments the count and a "down" arrow indicates that the program decrements the count.

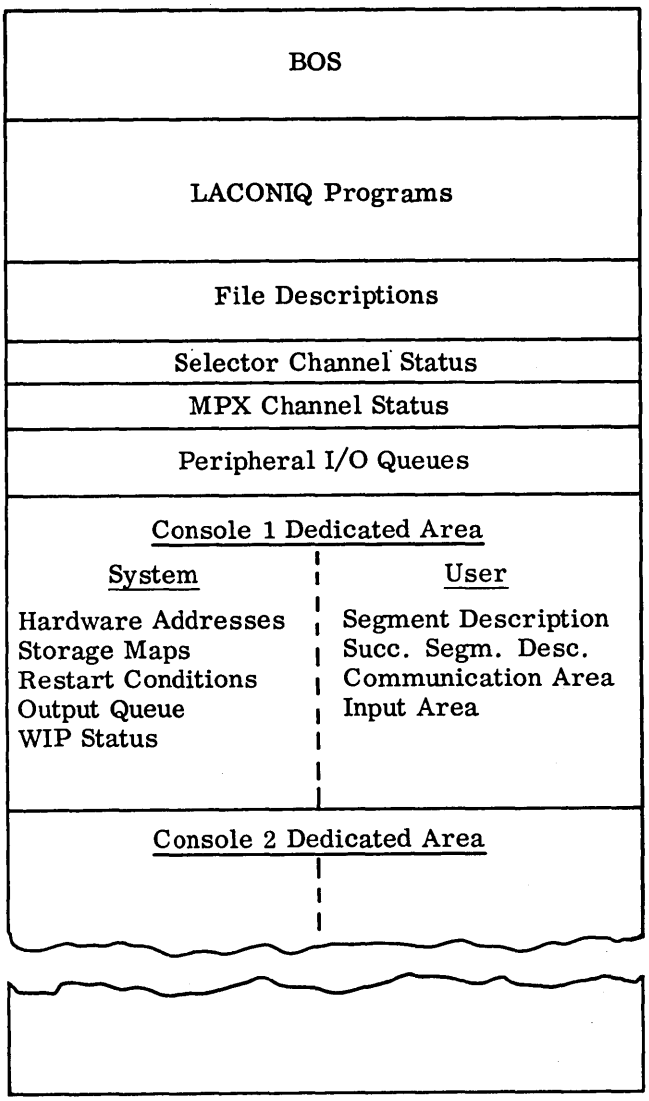


Figure 3—System area of core (two active consoles)

All indicators above the dotted line in Table I are individually interropated by the decision logic of WIPS. All indicators below the dotted line are checked as a group by WIPS to indicate work is still in progress for the console. The purpose of each indicator will become more apparent as we discuss the conditions which cause an indicator to be updated (in the remainder of

this section), and the interrogation of the indicators by WIPS (in the next section).

The following discussion applies primarily to local consoles whose console output requests are restricted to the CRT screen. There are minor variations for remotely located consoles and for output requests to the 1053 printer. The updating of indicators will be discussed in the order (top to bottom) in which they appear in Table I.

The console inactive bit is turned off when the initialization program reads in a card identifying the console as active. The bit remains off unless one or more of four things happen: (1) a catastrophic error occurs in application programs used by the console; (2) failure occurs in hardware used by the console; (3) the user fails to respond when input is expected;* (4) the user signs off.

Table I

CONSOLE STATUS WORD

Status Word Bit Identification	No. of Bits	Program Identification				
		WIPS	Console Scheduler	Console Interrupt	Peripheral I/O Interrupt	SVC Interrupt
Console inactive	1					
No. of console output requests	>1			↑		↑
Console output not initiated	1		0	1		1
Console page-back node point	1			0		1
Page-back not initiated	1		0			1
Console input requested	1			0		1
User has finished input	1			1.0		
Console input not initiated	1		0			1
Console input completed	1	0		1		
Peripheral scheduler needed	1	1.0				1
Tape output requested	1			0		1
Tape output not initiated	1		0			1
Segment input requested	1	1			0	1
No. of disk input requests	>1				↑	↑
No. of disk output requests	>1				↑	↑
No. of data cell input requests	>1				↑	↑
No. of data cell output requests	>1				↑	↑
No. of peripheral transfers	>1				↑	↑

If an application segment executes an SVC instruction for console output, the SVC interrupt program posts the "number of console output requests" and turns on the "console output not initiated" bit. Up to four requests may be stacked in each console queue to allow various combinations of writing modes (erase, write full-screen, and write with line-addressing). The "console output not initiated" bit signals WIPS to call the console scheduler which turns off the bit if the console control program successfully initiates output. The console interrupt program decrements the "no. of console output requests" and turns on the "console output not initiated" bit as each request is completed. When the number of console output requests reaches zero, the console interrupt program leaves the "console output not initiated" bit off.

The next two lines in Table I ("console page-back node point" and "page-back not initiated") are concerned with the capability for a user to return or page-back to a previous stage of processing, and restart from there. Each output display is optionally a page-back node point event. When WIPS detects the node point, the display and restart information is stored on disk. The user may invoke page-back by hitting a key on the console which has been assigned the page-back function.

The SVC exit routine turns on the "console input requested" bit and the "console input not initiated" bit if the segment description in the console-dedicated area indicates input is needed. When the user has finished typing his input and hits the ENTER key, the console interrupt program turns on the "user has finished input" bit. The console schedule turns off the "console input not initiated" bit when the console control program successfully initiates input. When input is completed, the console interrupt program turns on the "console input completed" bit. WIPS (later) turns off the "console input completed" bit when it enters a peripheral request in the peripheral queue for either the unique successor segment or the input analysis program (depending on whether the input is relevant to the selection of the next segment).

The "peripheral scheduler needed" bit is turned on by WIPS when a segment input is pending to ensure that the request will be recognized in case there is no peripheral I/O pending. (Recall that the peripheral scheduler would be called at interrupt time if there were outstanding peripheral requests.) The "peripheral scheduler needed" bit is also turned on by the SVC Interrupt program whenever data cell input or output is requested. (Data cell input or output requests require a unique successor segment to be input from disk so that core space will not be taken up while awaiting console input.) The "peripheral scheduler needed" bit is turned off by WIPS when the peripheral scheduler is given the CPU (provided the peripheral scheduler does not fail to schedule a request because of temporary shortage of free storage).

*Time-out procedures vary according to the application. For example, a console might be considered inactive if there is no response for 5 minutes in one application whereas a reasonable time-out for another application might be 30 minutes.

The "tape output requested" and "tape output not initiated" bits are handled in a manner analogous to console output.

The "segment input requested" bit is turned on by either WIPS or the SVC interrupt routine when a successor segment has been identified. It is turned off by the peripheral interrupt routine when input of the segment is complete.

The numbers of peripheral I/O requests are posted by the SVC interrupt routine and are decremented by one for each request completed.

The central scheduler

The logical flow of control for the central scheduler (WIPS) is shown in Figure 4 in simplified form.

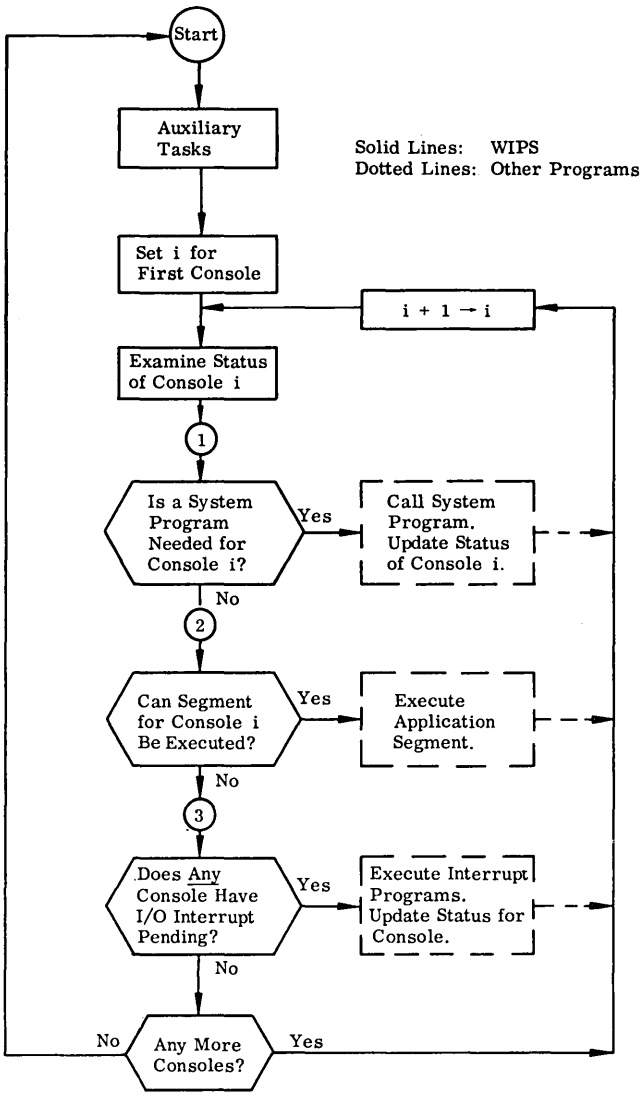


Figure 4—Basic functions of WIPS

The specific conditions which WIPS can recognize may be grouped into three categories.

- One or more LACONIQ systems programs are needed for input message analysis, output message scheduling, peripheral I/O scheduling, paging-back, etc.
- An application segment is ready for execution
- One or more I/O interrupts are pending

When and where to branch out of WIPS for the first condition is determined by examination of the 32-bit status word in the console-dedicated area. Each significant bit which is on in the status word relates to a specific need or event which must occur before an application segment for the console can be executed. When the need is satisied or the event occurs, the bit is turned off.

The needed action indicated by a bit (or bits) in the status word may be caused automatically as in the case of the I/O interrupts or by the calling of an appropriate system program by WIPS. When a bit(s) indicates that a system program needs to be called, WIPS branches to that program with appropriate parameters. If the system program performs its task successfully, return is eventually made to WIPS with the bit(s) off. If the system program cannot complete its task (e.g., because of temporary shortage of free storage), return is made with the bit(s) still on so that the need for the task will again be recognized during a subsequent pass through the WIPS loop.

The second condition, segment ready for execution, is recognized by WIPS when all bits in the status word are off. (An exception of minor importance occurs for remotely located consoles.) In the usual case, several passes will be made through WIPS before all the status bits are off. When the bits are off, WIPS will re-initialize (or initialize) segment conditions (e.g., restore base registers) and branch to the location in core containing the next (or first) instruction to be executed for the segment.

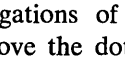
Satisfaction of the third condition, I/O interrupts pending, will occur when WIPS enables interrupts once in each cycle through the consoles. Interrupts are disabled in all other LACONIQ system programs (except for exceptional situations such as a fixed-length peripheral queue overflowing) so that there may be one or more I/O interrupt request stacked up. Upon enabling interrupts, the stacked requests will be handled by various LACONIQ and/or BOS programs. After updating of queues and the status words, return is made to WIPS where interrupts are again disabled and the checking of console status words is resumed.

When WIPS can find no on-line requirements to be met, the CPU is available for background. Background

processing will be interrupted periodically to investigate on-line needs. If no background is waiting, WIPS sets the CPU to the wait state. Entering the wait state without background processing is the current operating procedure. In the wait state, the meter used by IBM for accounting purposes does not run. A clock interrupt is used to periodically leave the wait state for WIPS to check for on-line requirements. Background processing will be implemented when the core size of the 360/30 is increased from 32k to 65k bytes.

Figure 5 shows the major logical decision flow in WIPS in considerably more detail. However, several auxiliary tasks are omitted from the figure for clarity. The tasks omitted are concerned principally with updating the priorities of requests in the peripheral queues which otherwise might not be serviced in a reasonable time period, procedures (which should rarely be executed) for handling a full queue condition, and general housekeeping functions. Entry point WIPS is entered only once, from the initialization program LACONI. After some additional initialization a pointer is set at WIPS1 for the first console. WIPS1 is reentered only from the WIPS program (the first decision branch after WIPS2) whenever the status words for all consoles have been checked and it is time to start over with the first console. WIPS3 is the reentry point from all interrupt routines and WIPS2 is the reentry point from all other system programs.

Assuming on-line requirements exist, flow proceeds from WIPS1 to WIPS2 where interrupts are temporarily enabled. The enabling of interrupts allows recognition by the system of the attention bit (triggered by the local user hitting the ENTER key) and allows other interrupt handling in cases where no application segments are in an executable state (where interrupts are always enabled). If an interrupt is pending, return from the interrupt routine is eventually made to WIPS3 which immediately branches back to WIPS2. Thus, all pending interrupts will be handled before the interrogation of console status bits proceeds downward from WIPS2.

The decision blocks in Figure 5 shaped like the following  are interrogations of the bits previously identified in Table I above the dotted line. As interrogations proceed downward from WIPS2, branches are made to the right either for additional interrogations or to call a needed system program. For example, if console output has been requested but not yet initiated, the console scheduler is called for output.

There is considerable significance in the order in which the console status bits are checked. (This order is the same as the bits were listed in Table I.) For example, it is impossible for a user at a console to wipe out pending console output or a page-back-in-progress procedure since these bits are checked before console input is allowed. The keyboard at the console is physically locked out until the pending requests are cleared.

The branches to other programs from WIPS2 occur as needed with eventual reentry at WIPS2 where the console pointer is incremented to allow interrogation of the next console's status word. If there are no system programs to be called and all I/O for the console has been completed, all significant bits in the status word will be zero. In this event, flow will pass straight downward from WIPS2 and the branch to WIPS4 will be taken. WIPS4 will restore the user's base registers (1 through 15), the storage protection key and the problem mode will be set, interrupts will be enabled, and a branch will be made to that location in core where the next instructions for the segment are located.

All I/O interrupts that occur during the problem state, i.e., during execution of an application segment, eventually cause WIPS to be reentered at WIPS3 where the decision branch goes to WIPS4. SVC interrupts of the exit type, i.e., those that request peripheral device or console I/O, that occur during the problem state eventually cause WIPS to be reentered at WIPS2. SVC interrupts of the non-exit type cause WIPS to be reentered at WIPS3 where the decision branch falls through to WIPS4.

CONCLUSION

Although little effort has been expended to measure the performance of LACONIQ (or even to select the best performance criteria), the system works well enough in the laboratory to proceed on an expanded system which will include background processing, peripheral device independence, additional facitilies for writing application programs, and compatibilty with the IBM Disk Operating System.

The major aims of this paper have been to point out the types of events in a system such as LACONIQ, the interfacing of executive-type programs, and one way that status indicators may be defined, updated, and interrogated. The methods used for the latter appear to be sufficiently efficient for use in the expanded LACONIQ system currently under development.

ACKNOWLEDGMENTS

The design and programming of the LACONIQ routines which have been described involved all members of the LACONIQ group. In addition to D. L. Drew and A. Reiter who have been previously cited,

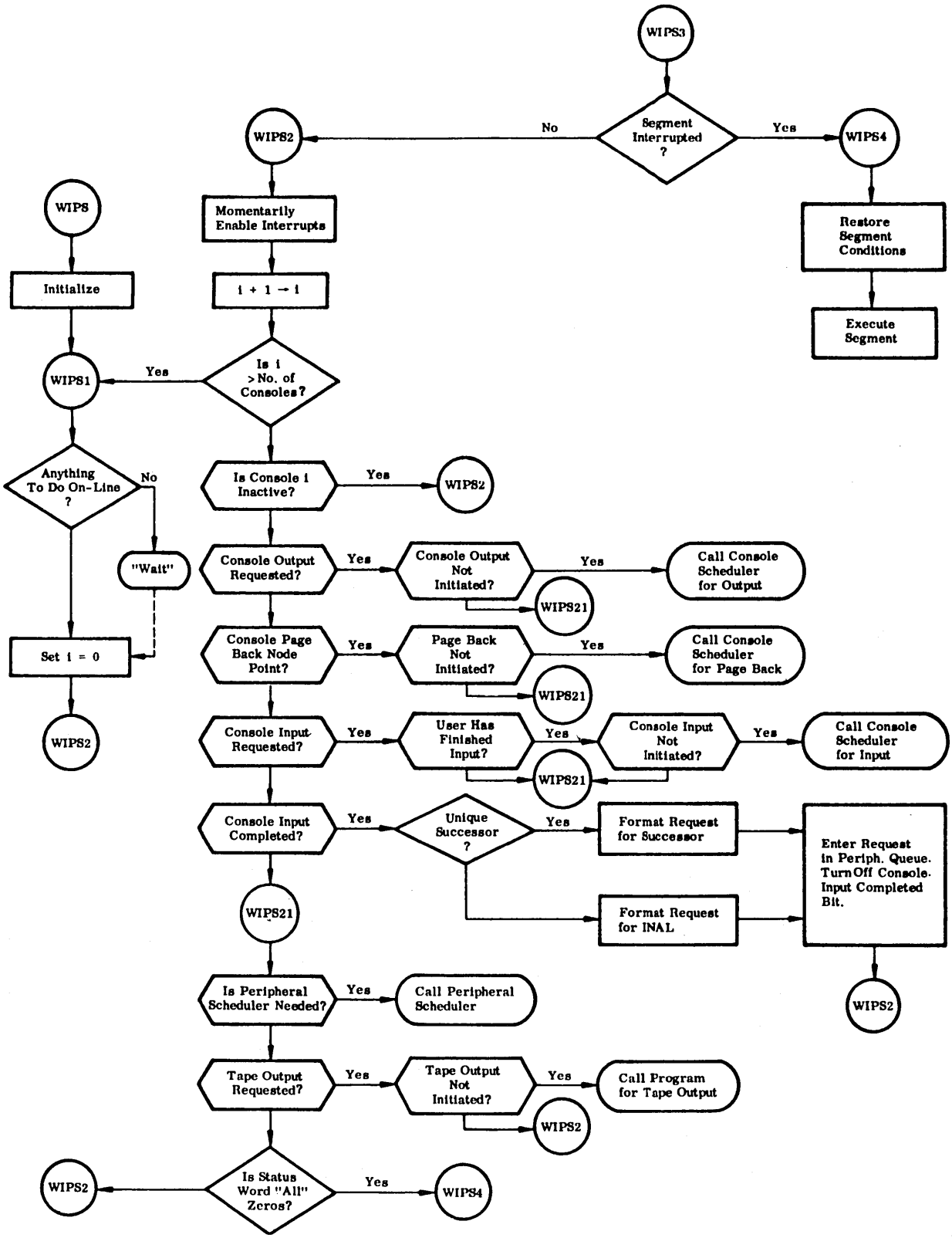


Figure 5—Logical decision flow of WIPS

the members of the group have included S. Burr, E. R. Estes, J. L. Fick, K. R. Gielow, A. J. Nichols, S. Shayer, and G. T. Uber.

REFERENCES

1 D L DREW
The LACONIQ monitor: time-sharing for on-line dialogues
Communications of the ACM (to be published)

2 IBM CORPORATION
IBM System/360 Summary Form A22-6810

3 M R STARK
ALTEXT-multiple purpose language manual 6-75-65-15
Lockheed Missiles & Space Company 27 Sep 1965

4 D L DREW
Two reference-retrieval dialogues
Information Retrieval Note 78 11 Aug 1965 Internal publication of Lockheed Palo Alto Research Laboratory

5 A REITER
A resource allocation scheme for multi-user on-line operation of a small computer
Proceedings of the 1967 Spring Joint Computer Conference May 1967

BIBLIOGRAPHY

R G CANNING *et al.*
Five approaches to the same data base problem
Proceedings of the Second Symposium on Computer-Centered Data Base Systems at System Development Corporation Dec 1965 available from Defense Documentation Center/Defense Supply Agency

J B DENNIS E L GLASER
The structure of on-line information processing systems
Information System Sciences Second Congress Nov 1964

J W FORGIE
A time- and memory-sharing executive program for quick-response, on-line applications
AFIPS Conference Proceedings Vol 27 Part 2 1964 pp 127-139

INFORMATICS, INC. Staff
Implementation procedure for on-line systems
Technical Paper TP-63-12-WHG

J M KELLER E C STRUM G H YANG
Remote computing—an experimental system Part 2: Internal design
AFIPS Conference Proceedings Vol 25 1964; 425-443.

H A KINSLOW
The time-sharing monitor system
AFIPS Conference Proceedings Vol 26 1964; 443-454

G E PICKERING E G MUTSCHLER G A ERICKSON
Multicomputer programming for a large-scale real-time data processing system
AFIPS Conference Proceedings Vol 25 1964; 445-461

J F SPITZER
The Colingo system design philosophy
Information System Sciences Second Congress Nov 1964

Appendix A

MAJOR DESIGN CONSIDERATIONS

(Excerpts from reference 1)

INTRODUCTION

The basic requirement in the design of the LACONIQ* monitor was that it should facilitate the programming and operation of on-line "dialogues." These dialogues typically consist of inquiries directed to a large data file, to which prepared programs respond in an attempt to help the system user find and display specific information. The prototype application is that of document reference retrieval.

This approach led to an event-driven monitor, as opposed to the usual "clock-driven" time-sharing monitor in which application program execution is carried on during a fixed interval, then interrupted until the program has another "turn." In the LACONIQ monitor, each application program is processed to completion. (This is only practical, of course, because the programs are limited in their duration.) Each such program corresponds to at most one step in the dialogue. Long operations might require that a sequence of such short programs (called program segments in the following descriptions) be executed, and this is foreseen in the monitor. The basic event that governs the timing of happenings in the monitor is then the completion of processing of a program segment.

This decision to set an upper limit on the amount of processor time each segment is allowed rather than to interrupt the segment ("roll it out" to a peripheral storage unit, and bring it back for later continuation) was based on a tradeoff between constraining the application programmer and increasing system response time. Given the conversational nature of the foreseen applications, it was felt that limiting segment execution time would not be a serious constraint because very few segments would approach this limit. (This has turned out to be the case.) The improvement in system response due to this doctrine derives not only from the smaller number of disk references but from the simpler mechanisms needed to handle the segments.

A second major design decision was to make the monitor poll the remote consoles, i.e., to accept input at the system's convenience rather than in response to interrupts generated by user's consoles ("contention" mode). Polling is made practical by the existence of adequate local buffers at the remote consoles. The principal advantage of this mode is that remote input

*LAboratory Computer ON-line InQuiry

can be scheduled, so no dedicated input buffer areas are required.

All the resources of the system—communication lines, core storage, use of the CPU, and use of the I/O channels—are scheduled in one way or another. The most fundamental scheduler, the work-in-progress scheduler, calls in both program segments and systems routines as appropriate and when needed.

The monitor was not primarily designed to facilitate programming at the user console, although incremental assemblers or compilers can be added in the form of applications. The basic criterion was rather to make it easy for a skillful programmer to prepare an application with which the (usually untrained) user, at his console, could interact in a language natural for that application. This monitor might be described as a three-level system, with the programmer mediating between the layman-user and the computer, and is a step toward making the power of the computer more generally accessible.

It is probably obvious, but perhaps worth mentioning that a further requirement on the monitor was that it should be capable not only of supporting several terminals processing a given application, but also several different simultaneous applications.

Many of the detailed features of the monitor are directly related to the IBM 360, and some even to the configuration of the particular computer system at hand, which consists of a 32k 360/30 main-frame with two 2311 disk drives, a 2321 data cell drive, a tape unit, and multiple remote CRT consoles (IBM 2260 and Sanders 720). The disk drives each store 7.2 million bytes with average access time of 85 msec, the data cell stores 4.8 million bytes with a maximum access time of 600 msec. The CRT consoles display, respectively, 960 and 1024 alphanumeric characters at 120 characters per sec (line rate). Some hardware factors which have been important in their influence on monitor design are the small core storage, the large peripheral storage, and the local buffers for the console displays.

The first version of LACONIQ was operational in December 1966, supporting three IBM 2260 consoles and using two disk drives and a data cell. It has been continually upgraded and at this writing it is being set up to have a high degree of compatibility with the IBM Disk Operating System and OS/360. The changes include exploiting a larger core storage (64k bytes) and introducing a background capability. Depending on the size of the background partition and the application mix it is foreseen that the monitor will support between six and twelve CRT consoles with an (approximate) 2-second response time. System overhead is estimated at 25%, and it is believed that more careful measurement will show this to be pessimistic.

BIBLIOGRAPHY (From reference 1)

"Programming Real Time Systems," by James Martin, Prentice Hall, 1965, is an excellent general discussion of the problems encountered and dealt with in the SAGE, Project Mercury, SABRE, PANAM, New York Stock Exchange, and other systems.

Information on the well-known Project MAC is available in Memoranda, Technical Report, and Progress Report form from Project MAC,, Massachusetts Institute of Technology, Cambridge, Mass.

Appendix B

TYPICAL SEQUENCE OF EVENTS

Consider that two consoles are currently being used on-line and both are part-way through some retrieval process for their respective data files. Core might be allocated as shown in Figure 6. Assume that there are presently no outstanding I/O requests for either console and the segment for console 2 is in execution. In the hypothetical and simplified sequence of events that will be described, abbreviated names for the major system programs and subprograms will be used as follows:

Central scheduler	— WIPS
Peripheral scheduler	— PIOS
Console scheduler	— RIOH
Console control	— LINECT
Peripheral control	— PERIF
Input analysis	— INAL
Interrupt programs:	
SVC	— SVCROU
Peripheral I/O	— PINT
Console output	— KOP
Console input	— RIP

Sequence of events:

1. Segment 2 executes an SVC requesting disk input.
2. SVCROU places the input request in the peripheral I/O queue and transfers to PIOS.
3. PIOS finds the request in the queue and the device free. PIOS calls PERIF.
4. PERIF initiates the input commands for the requests and marks the device busy. PERIF returns to PIOS which returns to WIPS. m
5. WIPS checks needs for the next console (console 1 in this case) and finds that segment 1 can be executed. (Segment 1 has presumably been waiting

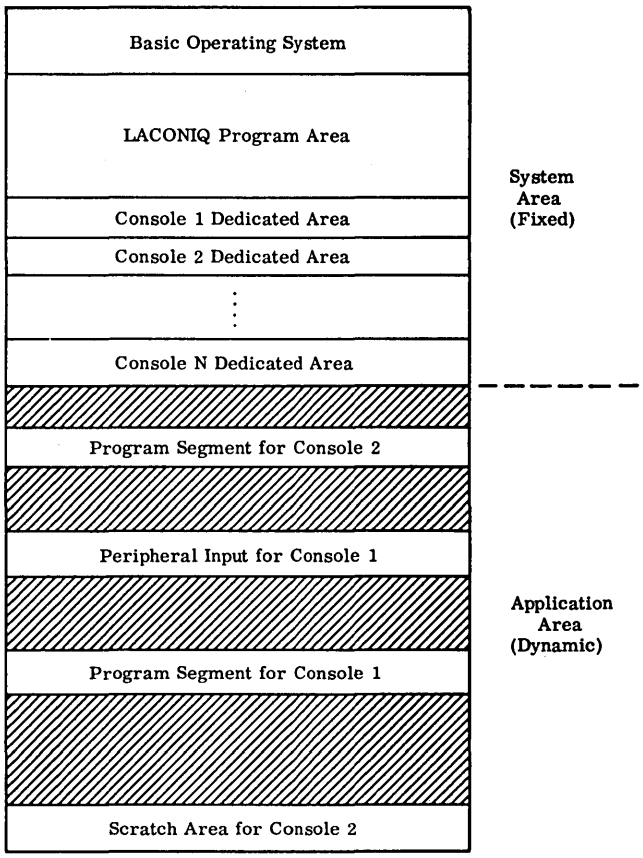


Figure 6—Typical items in core for two active consoles

for data input from peripheral which has been completed.)

6. WIPS gives control to segment 1 which manipulates its data and requests output to peripheral.
7. The SVC which segment 1 executed for an output request is interpreted by the appropriate SVC routine. SVCROU formats the request and places it in the peripheral queue and transfers to PIOS.
8. Steps 3 and 4 are repeated for the output request. WIPS finds that there is nothing to do for either console and "WAITS." (Background will eventually be performed in this situation.)
9. The output completion interrupt occurs for segment 1. PIOS is called to initiate another request, but none are pending so PINT is called immediately to finish processing the interrupt.
10. PINT posts appropriate bits for later interrogation by WIPS, re-sets indicators for hardware status, and goes to WIPS.
11. WIPS, which had been "waiting," finds the output for segment 1 completed and puts segment 1 into execution again (at the point in the segment immediately following the recently completed output request).
12. An interrupt for completion of the input for segment 2 occurs while segment 1 is in execution.
13. PIOS finds no more requests in the queue and calls PINT.
14. Status indicators are set by PINT to indicate that the input for segment 2 is complete. PINT returns to WIPS.
15. WIPS allows segment 1 to resume execution at the point interrupted.
16. Segment 1 requests console output. Assume the segment description for segment 1 requires that a user decision (console I/P) be made before processing (with another segment) can continue.
17. RIOH finds the line is free, formats the request, and calls LINECT which executes commands to put the segment 1 display on the screen. LINECT returns to WIPS.
18. Segment 2 is put back into execution by WIPS.
19. The user at console 1 types in his input and hits the ENTER key. (It is very unrealistic that the user could respond as rapidly as indicated here, but the situation is satisfactory for illustration.)
20. An interrupt routine in LINECT posts that input is waiting in the buffer at console 1 and goes to WIPS.
21. WIPS calls RIOH which calls LINECT to initiate commands to pull in the buffer contents. Return is to WIPS.
22. WIPS puts segment 2 back into execution.
23. The input from console 1 is completed causing an interrupt.
24. RIP turns on a "console input completed" indicator and returns to WIPS.
25. WIPS calls INAL to process the console input.
26. INAL analyzes the input and determines the next segment to be executed for console 1. The input request for this segment is placed in the peripheral queue by WIPS.

etc.

An executive system for on–line programming on a small–scale system

by LANCE V. MOBERG
Univac Federal Systems Division
Sperry Rand Corporation
St. Paul, Minnesota

INTRODUCTION

The role of Executive Control Programs is increasing with the number and size of the applications of computers to data processing problems. As problems become more complex and as the computer's external equipment becomes more numerous, Executive Control Programs serve as monitors in increasing the efficiency of the computer as well as its capability of shared utilization of hardware and software. They are, therefore, an enlightened compromise of hardware and software.

Executive Control Programs can vary from simple to complex; from use on small system configurations to large-scale, costly systems. They can be simple, interrogate-and-jump routines; they can be completely program-encompassing; or they can fit almost anywhere along the range between these two.

The system described in this paper represents one of few currently operational data processing complexes bridging the gap between small, limited budget configurations and very large and costly systems. This gap is perhaps more apparent in the militarized market than commercially. This militarized system fills a need for a small-scale configuration with nearly unlimited capability for solving extremely large problems. Primary use of the system is for information retrieval and computational assistance. However, the capability for on-line programming is a most significant feature of the support software.

System programs developed include the Executive Control Program, a machine language assembler, CS-1 Compiler, JOVIAL Compiler, and utility routines. This paper is primarily concerned with describing the Executive Control Program. Software objectives in this development were in the interests of implementation rather than towards a technological breakthrough. Methods employed are largely adaptations of already known programming systems and techniques. Major distinction in this effort is that features of many systems were combined into one system resulting in an operational product. Hardware objectives used slightly different and, in some cases, completely new approaches to supporting the software and system users.

The true objectives must be viewed from a total system standpoint. These are best conveyed by the question asked of Univac at the project's outset. Can a small-scale computing system be designed, developed and delivered which provides real-time response to a minimum of sixteen independent consoles manipulating a common, large data base and ensure complete system integrity?

This meant that priority control, time-sharing, real-time demand operation, and complete program and data security were among the many obstacles. In addition, the software system must support the illusion of infinite computer memory. This meant maintaining complete flexibility such that an operator untrained in computing could connect and disconnect functional elements, at will, to solve his problems. This was considered a form of an implicitly programmed system.

After some background information is presented, the remainder of this paper describes the extent to which these objectives were met.

Active hardware components

Central computer of the system is the UNIVAC 1218; a member of the general-purpose 18-bit word length family of the Univac Federal Systems Division product line. The computer has a 32K word memory with six microsecond access time and eight input/output channels. Time-sharing features were incorporated to provide privileged/non-privileged instruction modes, memory protection, and time accounting clocks.

The mass storage unit is a Data Products Corporation model 5025 Disc file with a total storage capacity of 400 million bits. The unit consists of sixteen separate discs each with an independently moving arm containing heads to access eight tracks from one position. Average access time is 125 milliseconds.

System consoles are Raytheon CRT units with UNIVAC designed control circuitry and internal core

memory. A message at the console consists of a maximum of 34 lines of 80 alphanumeric characters each. Extensive editing can be performed at the console, by use of the keyboard special controls, independent of computer action. The console interfaces with the computer through a buffer unit where each connected console has storage for 39 full messages. Messages stored in the buffer are called and returned by means of keys on the console. In addition, a TO COMPUTER key and twenty-four program select switches signal the computer with encoded interrupts. Seventy-four indicators are also present on each console for program run lights and message identifiers. Figure 1 shows a typical system console.

Operational characteristics

Hardware and system software interact to provide many conveniences for the programmer and user of the system. The first of these is a two-stage priority control. Programs are executed based first on the importance of their function, and secondly on the console from which they are requested. There are six levels with which to specify function importance; console priorities are relative to each other and changeable. Function (or program) priority is a significant factor in controlling program swapping to the disc file.

Areas on the system disc file are identified and referenced by unique, programmer contrived, alphanumeric names. One-third of the file is catalogued in 4096-word segments addressable in 512-word blocks; the remainder (the data base) is referenced as one area with a unique alphanumeric name and addressable in 512-word blocks. All disc file references are subjected to access permissibility tests established from the out-

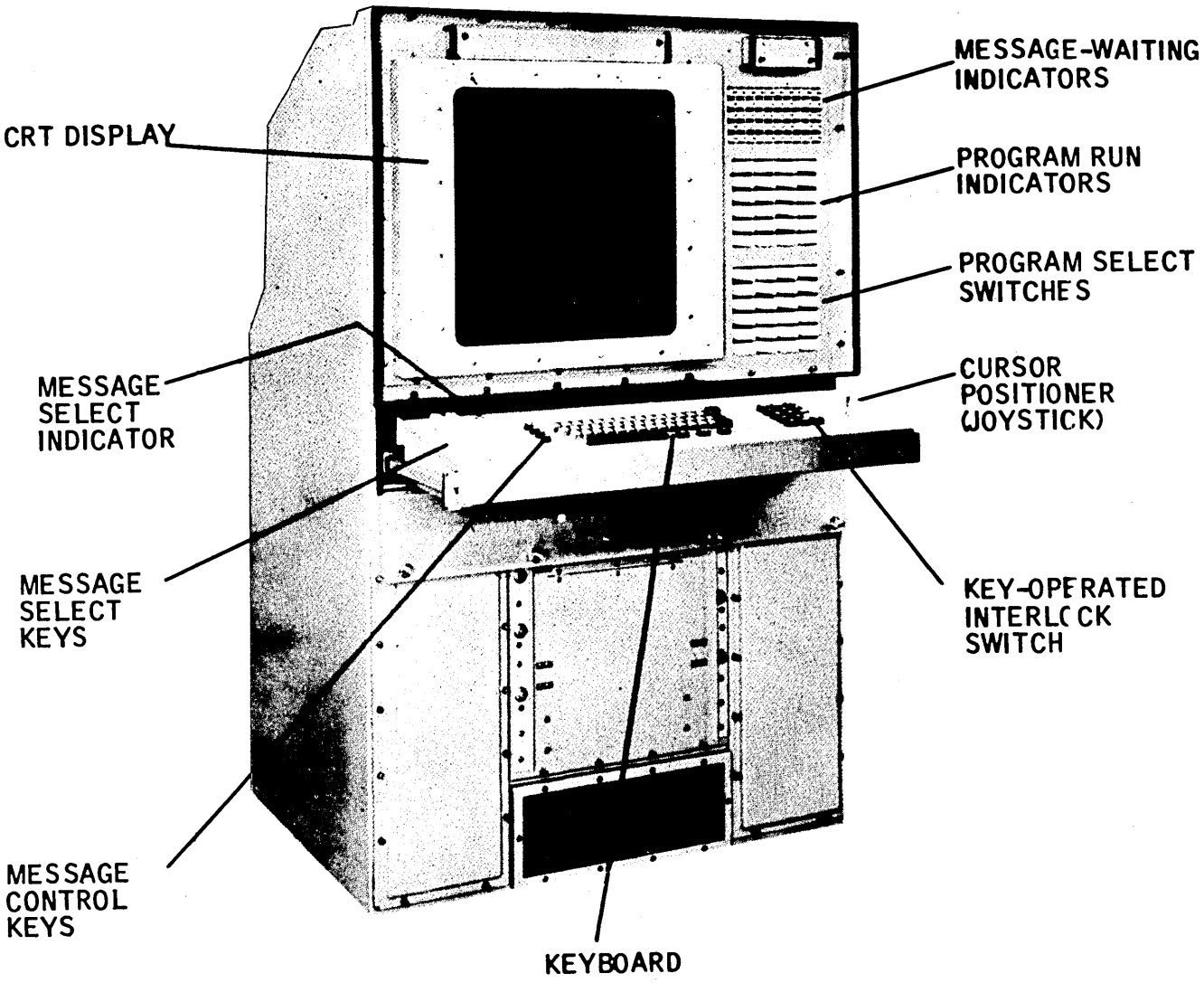


Figure 1—System console

side (site manager). Areas are categorized as temporary, permanent, or console storage.

The system consoles are identified by logical unit assignments. Program-to-operator communications can be by simple button depression and/or definitive messages.

Library programs can be dynamically linked together in variable sequences to provide an automatic schedule of programs to execute. Schedule length is completely variable and has no practical limit. For example, on-line programming can be scheduled by linking the Source Language Librarian, language processors, supervisor communication program, and the worker programs under development. All system language processors could be included in the schedule (sequence) with the specific processor desired being selected by a single button depression during execution of the schedule.

System support software

System software consists of an assembly system, two compilers, source program library and librarian, object program library, utility system, and worker programs operating under control of the Executive Control Program.

All programs operating outside of and under the control of the executive are termed "operational programs." An operational program may consist of one program or a sequence of programs, each individually designed to accomplish a particular job. Therefore, an operational program is otherwise termed an Operational Sequence of JOB programs. The assembler, compiler, source language librarian, and utility system are examples of support programs that fall into operational program classification. Operational programs are constructed by linking together a sequence of JOB programs generated through the assembly or compilation process. Typical problem solving JOB programs are mathematical routines, computational-assistance programs, data acquisition algorithms, data reduction and formating routines, and report generators.

Operational programs are entered into the system in the following manner: Any operator in the system may, from any console, enter JOB programs coded in source language into the source library via the source language librarian. Source language may be prepared and entered via the console display or punched paper tapes. Having entered his JOBs in the source library, the operator may then call on a language processor to generate his programs placing them into the object library. Each JOB program is assembled or compiled separately.

System control is under complete direction of a site manager trained in the capabilities and operational characteristics of the software. Among the duties of this site manager are security control, program library and data base management, and system utilization factors such as priorities and execution limits. One of the consoles in the system is dedicated to these duties and use by the site manager. Operators may communicate with the site manager by transmitting messages between their consoles and the Supervisory Console.

With all JOBs to comprise an operational program in the generated library the operator may then request the site manager to construct an Operational Sequence. A set of worker programs, requestable from the Supervisory Console, aid the site manager in this task. These worker programs are termed "System Supervisor Functions." In constructing an operational program, the operator supplies the site manager with certain information and he in turn, dictates operating constraints. Information supplied to the site manager is:

1) Identification of JOBs to form the operation program and the sequence in which they are to be executed,
2) System data areas referenced by each JOB,
3) Hardware facilities required by each JOB, and
4) Console program select button assigned to activate the sequence.

The site manager specifies the priority level at which the sequence should run and the execution time. Following accumulation of this information, the site manager constructs the Operational Sequence by executing a "System Supervisor Function" for entering this data. A second "System Supervisor Function" is then executed to submit authorization to the executive to allow the new sequence to manipulate its declared system data. The operational program is now ready for execution by the operator seated at his console.

Program construction

The system language processors produce programs in page format. Pages are 512 words in length. This size is the common denominator resulting from the computer instructions address domain and the special memory interlace modification which provides program protection. As a processor generates, instructions are positioned at the top of a page progressing downward, while local data and indirect addresses are positioned in a transfer table located at the end of the page progressing upward. The processor automatically generates indirect addressing through the transfer table thus producing a free-standing set of instructions requiring no modification for relocatable loading.

The CS-1 and JOVIAL Compilers in addition to constructing pages optionally generate a program to execute interpretively. In this way a programmer may write a program in high-level language with no consideration for the size of the generated program. A

special routine is automatically generated by the compiler to interpret all memory references external to that page and call the proper page into memory.

A control block for executive linkage is also produced by the language processors. It is by means of this control block that the programmer can reference disc file areas symbolically. Two directives are used by the programmer to declare data areas; these are PERManent and TEMPorary. Alphanumeric names coded in the operand positions following the PERM and TEMP operators are automatically assigned "name-numbers." The processor records the permanent alphanumeric data names in the control block to enable the executive to perform access validation when executing the program. The name-numbers are employed in the generation of executive calling sequences for PUT and GET requests for data on the disc file. Hence, the programmer simply writes GET • BOX 1A(3) • BUFFER having declared the area by writing PERM • BOX 1A.

Permanent data areas are catalogued in a system directory by the site manager. Data contained within these areas are preserved by the executive until removed by the site manager. Access controls imposed by him are honored by the executive to provide additional system integrity. Accessed data or instructions can be validated at load time and/or requested dynamically during the program's execution.

Temporary data areas live only during the execution of the declaring program. Provisions are included for passing these areas between jobs of an operational sequence and for equating their symbolic names. Temporary areas can be obtained by pre-declared requirements satisfied at load time and/or dynamically during execution.

A third type of data area can also be accessed as instructed by the console operator. These areas are obtained and released on request from the operator and can be accessed by any program activated at his console. After generating a program, the language processor can be instructed to enter the object language into the generated library on the disc file. At this time the program is assigned a JOB ID which is presented to the operator for future reference.

Operator capabilities

The system's consoles contain, in addition to CRT and typewriter keyboard, a variety of indicators and switches. Each console contains a bank of 24 interrupt producing switches (with accompanying code) and a bank of fifty indicator lights which can be illuminated by the computer. As described in preceding paragraphs, the site manager assigns program sequences to the console switches. For example, the assembler, compiler, and source language librarian would be assigned to a console to be used for on-line programming. The dedication of these switches to these functions is recorded in a system table until the site manager rededicates the switches to other functions.

The operator activates program sequences by depressing the appropriate program select switch. A typical sequence of events following a program request is:

1) Executive acknowledges receipt of the request, loads and initiates the program
2) Program queries the operator with a message
3) Operator types a message and transmits it to the computer
4) Executive stores the message in the requesting program
5) Program proceeds as directed.

Other operations available to the operator at his console are:

1) Direct the execution of this program by button depressions in lieu of or in combination with messages
2) Turn off indicators
3) Edit messages via character, word line or message errasures, inserts, or copy
4) Store up to 39 messages in auxiliary storage in the console buffer unit
5) Any other function available through assigning program sequences to select switches.

Operations available for on-line programming would also include inspect and change procedures for program and data, program dumps, request for mass storage area, and various data transfers and code conversions.

The exact operation of the console is not relevant to this paper; what is important is the flexibility programmed into the system to allow changing the mode of any console. An on-line programming mode was chosen for this discussion. Neither the system or its consoles need be dedicated to this mode. Modes are changeable by the site manager at any time with any desired frequency. In examining how this applies to on-line programming the following areas are of interest. If a programmer (temporarily termed operator) desires to compile a "disjointed segment" of a program and execute the results he may do so by the following operations at his console:

1) Enter the segment source language into the system source library. This can be done by console typewriter entry, if so desired, or by loading paper tape.
2) Generate the object program by executing one of the language processors.
3) Before recording the object program in the object library, inspect the generation on the CRT. Diagnostic warning codes pinpoint syntactic errors.

4) Produce a hard copy, if desired.
5) Correct the program source language and regenerate the object program if necessary. Minor generation errors could be left for patching at execution time if desired.
6) Finally, enter the object program into the system library.
7) Request the site manager to assign this new program to the console.
8) Initiate execution.

Attributes of the executive and language processors which make this possible are many. Program page construction provides independence from execution area. The memory area in which the segment executes now may or may not be the same as the area for execution of the segment when it is joined with its parent program. Breakpoints allowing operator (programmer) intervention are possible by program request to the executive. At this point data can be inserted into disc areas or core memory for processing by the segment. Program execution will not proceed until directed by the operator. Execution by the executive is made possible by the Operational Sequence of JOB programs concept. Since the only definition of JOB is any set of instructions produced by a single run of a language processor, disjointed segments would be considered JOBs. In fact, one or more disjointed segments could be executed in combination (sequentially) by constructing an Operational Sequence of them. Communication between segments can be via disc file areas or operator console.

Naturally, not everything is automatic with this system. Certain attention has to be paid to system philosophies when designing programs. As in any system, awkwardness can develop if planning was not sufficient. However, the majority of the problems of on-line programming can be met with this system. This includes heretofore unmentioned capabilities of debugging aids, executive diagnostics, and backup library.

Standard supervisory functions provided by the executive for exclusive use by the site manager are enumerated below.

- Create operational sequence
- Enter data names and job list
- Abort operational sequence
- Designate facility status
- Core memory print
- Core memory inspect and change
- Disc file inspect and change aid
- Magnetic tape inspect and change aid
- Enter time-of-day and date
- System runout

Executive control program

The Executive Control Program has four main functions:

- Control of all Input/Output
- Receive, interpret, and process all interrupts
- Maintain communications with each console operator and schedule his requests for execution
- Allocate mass storage and control access to the unit

The heart of the executive is the switcher. All executive routines relinquish control either directly or indirectly to the switcher, and the switcher in turn routes control to the appropriate executive routine. This procedure is based upon a periodic scan of six queues of "outstanding work." A real-time clock is preset with a value sufficient to interrupt the computer at an optimum interval to cause the switcher to interrogate these queues. They are, in order of scan:

- Unanswered interrupts
- Executive lost control points
- Executive ready return points
- Executive task timing queue
- Executive task queue
- User programs

If no work is indicated by any of these queues, the executive idles until a real-time clock or other interrupt occurs.

At the time of an interrupt, the hardware automatically inhibits all further interrupts from disrupting a routine. The answering routine thus has a chance to perform necessary housekeeping chores such as saving register contents and setting up a return point in a queue before releasing the interrupt inhibit. Since an interrupt causes routines to be suspended until the interrupt is processed, the Unanswered Interrupts queue carries the highest priority. Unanswered interrupts are interrupts that have been received, but not yet analyzed because the interrupted coding was not suspendable. After an interrupt has been processed, the switcher determines if any other interrupt answering routine was suspended, and if so, routes control to perform the analysis.

If the switcher finds that there are no unanswered interrupts, the Executive Lost Control Point queue is checked to determine if interrupts have taken control from the executive. If so, the necessary registers are restored and control is returned to where the executive was last in control. The switcher next checks the Executive Ready Return Point queue. When the executive uses the special REXEC instruction to call upon another executive routine to perform an executive func-

tion (such as a disc file read or write), control is relinquished pending completion of the operation. When the operation is complete, an executive ready return point is created. Executive processing is expedited by returning to ready points before considering other places to go.

The Executive Task Timing queue is metered by the real-time clock routine in conjunction with the real-time clock interrupt. Executive tasks which are to be initiated upon expiration of an interval of real-time are recorded in this queue. Examples of this are delayed turn-off of a console indicator or periodic attempts to execute routines waiting for some other dependent but separate action. Tasks revealing expired time intervals are given control at this point. Newly received requests for executive action are queued in the Executive Task queue. The last attempt to route control to an executive routine is accomplished by a dispatcher scan of this queue. If no work is to be done in the executive, the dispatcher checks the processing priority table (PPT) to determine if any job program is to be executed. If so, control is routed; if not, the executive becomes idle. Jobs are entered into the PPT table by the request processor. Console requests for job executions are obtained by the request processor from the Request Table. An attempt is made to queue the request processor at each real-time clock interrupt. If the request processor discovers new jobs to be initiated, it in turn queues the scheduling processor.

The scheduling processor determines which job(s) the program loader should load into core from the disc file. The considerations for scheduling a job are in descending order:

- Job priority
- Eligibility
- Available core
- Availability of facilities, if required

If necessary, currently resident jobs are swapped to make room for higher priority jobs. After a job is loaded, control is returned to the switcher which in turn gives the job control.

There are three other major executive routines in the Executive Control Prorgam:

- Terminate
- Indicate
- Overtime and overdue clock routines

When a job terminates, "terminate" performs housekeeping for the operational sequence of jobs. Terminate may also queue the scheduling processor to initiate the next job if the terminating job was not the last one in the operational sequence.

Indicate, as its name implies, transmits information from the executive to each console. Job status, system status, sequence progress and all error conditions are communicated to the operators.

The overtime clock routine causes a job to be aborted if the job exceeds its maximum allowable execution time. If a job is eligible to be loaded (either new or swapped) but encounters delays, the overdue routine updates its priority at intervals to ensure that it will be loaded without excessive delay.

Figure 2 depicts the general processing flow of the executive.

The operation of the executive is completely based upon the proper coordination of its various modules such that they come in and out of control in the proper sequence to provide the necessary processing. Modules are constructed to accept tabular information, from one or more tables, as their sole input and, as a result of their processing, produce tabular information as their sole output. Control is routed through the executive in one of two ways: queuing of a module, or requesting an executive process. Processes are identified to the Executive Control Coordinator (ECC) by means of an ECC Flag. A program may request these services of the executive:

- Put data on disc file
- Get data from disc file
- Query/inform operator
- Convey message to program
- Read message from originating console
- Check request completion
- Convey button code to program
- Read Program Relocation Register (AMR)
- Set AMR and transfer control
- Set AMR, transfer control and save return ponit
- Facility request/release
- Console Buffer I/O operations
- Printer I/O operations
- Magnetic Tape operations
- Paper Tape operations
- Segment load
- Obtain temporary disc file storage area
- Validate access to permanent disc file storage area
- Validate access to console storage on disc file
- Inter-interlace data move
- Terminate

The executive contains extensive error checking and parameter validation procedures to ensure that neither the operator nor the user program can destroy the system. For those conditions which are not detectable, the system is designed to turn their effects back to the erring program, thus maintaining the integrity of the executive as well as other user programs.

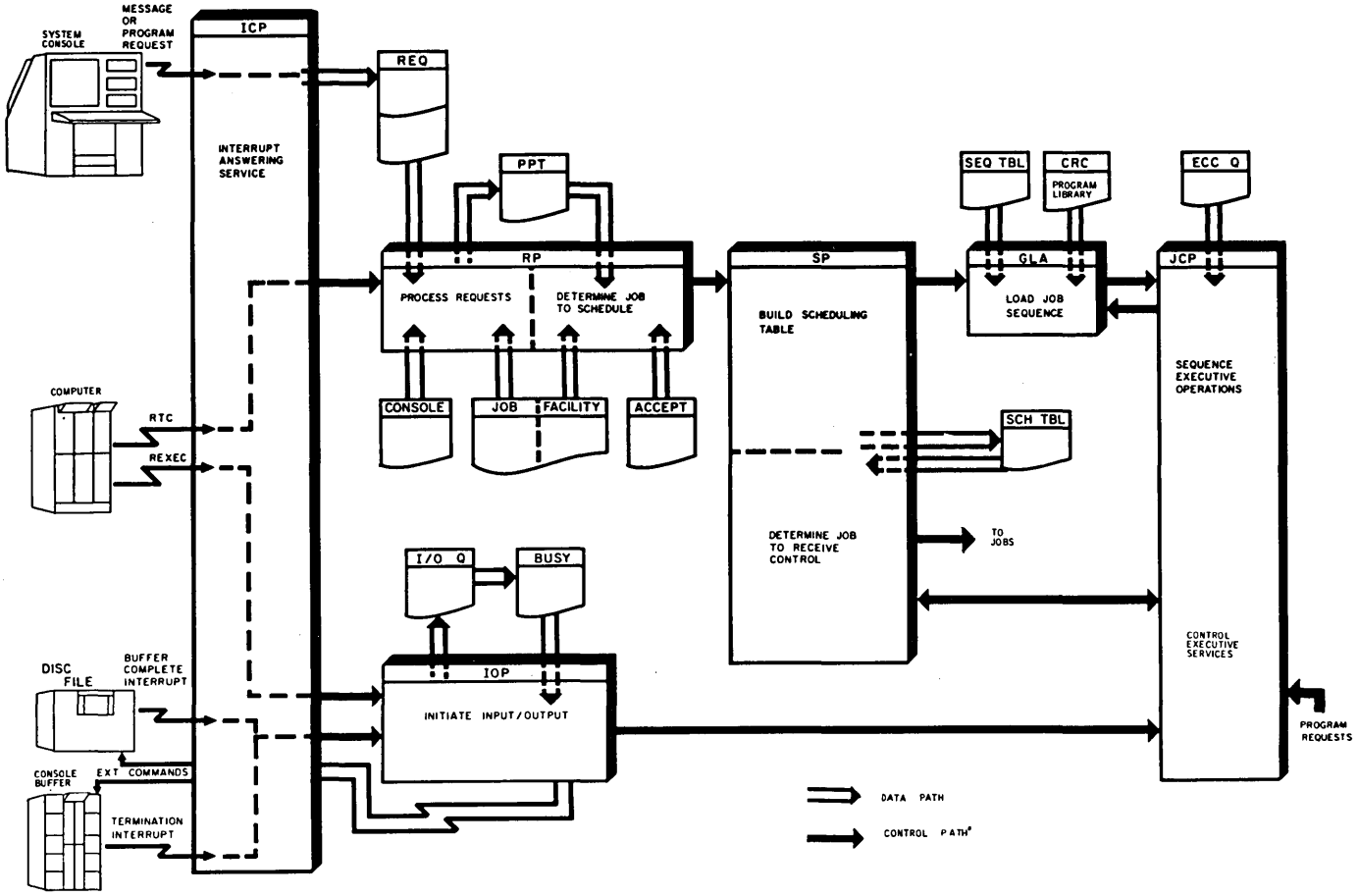


Figure 2—Executive control program flow diagram

Use of the magnetic tape, paper tape, and printer facilities is controlled by peripheral to program assignments. Provisions are incorporated for queuing requests for facilities already in use and aborting of programs requiring facilities rendered inoperable for servicing.

Input/Output

The three main Input/Output modules of the executive are the Input/Output Programs (IOP), Input/Output Coordinator (IOC), and the Interrupt Control Programs (ICP). Basically, the IOP consists of handlers and parameter validation routines, the ICP consists of interrupt answering and evaluation routines, and the IOC consists of those routines used for general housekeeping functions and for routines used to coordinate and interface the IOP and ICP.

Processing in the input/output section is based upon the simple approach. While some systems may require more complex algorithms, this approach proved practical for this system. Of primary consideration in choosing this philosophy is the basic data flow of the operational system. Only two devices are in the primary line of flow; operator console and mass storage. Considering the transfer rates, data construction, non-linear files, and response times, the following simplifying ground rules were established.

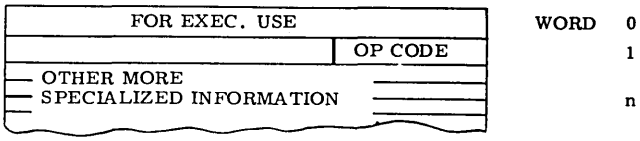
1) In determining what action to take next, input/output should be considered first;
2) Job priority need not be applied to I/O servicing; and
3) Swapping programs with outstanding I/O operations would not be economical.

As requests for input/output operations are received by the executive, they are automatically queued in one of the Input/Output Handler Queues. There is a separate queue for each handler, i.e., for each piece of peripheral equipment. The length of the queues are set, based upon the estimated usage of the associated equipment. The queues contain the address of the parameter packet of the requested operation.

Input/Output operations are initiated whenever a channel becomes "not busy." Associated with each channel is a channel busy flag which is zero if the channel is free to use and set to the address of parameters of the requesting program if the channel is busy. At request time, this channel busy flag is examined. If no other requests are queued in the I/O queue and if the channel busy flag is clear, the request is initiated, otherwise, the request must work its way up the queue.

Each time an interrupt is received the Input/Output Active Flag (IOAF) is referenced. This flag is used to indicate that status of the entire input/output section. If upon receiving an interrupt this flag is clear, immediate interrupt status word evaluation proceeds and control branches to the proper evaluation routine. If this flag is set, i.e., if the I/O section of the executive is already busy, the interrupt status word(s) and a return point are stored into the unanswered interrupt queue. This queue is later scanned at the convenience of the executive, at which time normal status word recognition and evaluation is performed.

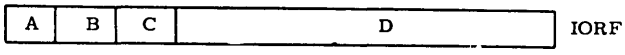
Programmer requests for input/output operations are always accompanied by a parameter packet with the desired control information. A special flag called the Executive Control Coordinator Flag (ECC) accompanies each packet and denotes the type of executive action required. The I/O packets themselves are of the following general form:



Word zero of the packet is reserved for executive usage and is referred to as the Input/Output Request Flag (IORF). As a request is processed, certain vital housekeeping and control information is placed in the IORF to enable the executive to maintain proper control over the request. The control information necessary for the executive's performance is:

1) An index to a table of vital registers
2) An input/output active indication
3) An input/output completion indication
4) An indication of the source of request

The IORF used in the MWRT executive of the form:



A = 1 I/O Has Not Completed
0 I/O Has Completed

B = 1 JOB Program Request
0 Executive Request

C = 1 Check of request's status received
0 Check of request's status not received

D = Must contain one of the following:
(1) Job program's table index (PPT index)
(2) Executive task number
(3) Address of working storage

The operation code for each function is defined to be the same for similar operations on the various pieces of peripheral equipment. For example, an operation code of 21 means a read operation whether on the disc file, the magnetic tape or any piece of equipment which might be associated with the Executive System.

Job control

The Scheduling Processor determines which job programs to schedule for loading. Considerations, in order, are:

1) Priority
2) New request versus initiated
3) Eligibility
4) Shelving (swapping)
5) Facilities
6) Sufficient core

A counter is used to determine if the scheduling processor should be queued. For each new request that is received the counter is incremented by one. After a job terminates or is aborted the counter is incremented by one unless that job was the last in the sequence. Each time a job is swapped, the counter is incremented by one. When a job is loaded the counter is decremented by one. This procedure ensures that every job will be scheduled to be loaded.

Six levels of program priority are considered by the Scheduling Processor. They are, in decreasing impor-

tance: Emergency, Crash, Priority, Interrupt, Working, and Deferred. In general, the priorities differ only in the power they have to usurp control. Table 1 summarizes these priorities.

Table 1. Summary of priorities

Emergency	Will destroy prevailing programs and pre-empt facilities to allow immediate execution.
Crash	Will pre-empt facilities and cause programs to be swapped to allow execution.
Priority	Second level capable of pre-empting facilities and causing programs to be swapped to allow execution.
Interrupt	Intervenes between job steps of lower priority sequences.
Working	Normal processing level of most programs.
Deferred	For infrequent, time-independent executions of the background variety.

Three tables are manipulated by the Scheduling Processor; Processing Priority Table (PPT), Link Table (LINKTB), and Processing Priority Link Table (PPLT). Each serves its own function in the system to: 1) maintain orderly records of all active requests (PPT), 2) maintain a priority ordered structure (LINKTB), and 3) expedite dispatching of control amongst the resident programs (PPLT).

After a job is scheduled for loading, the Program Loader is called upon to perform the necessary tasks to load a job into core from the disc. A Centralized Catalog Handler (CCH) is used by the program loader to obtain the location of and information about the program in the object program library on the disc file. All mass storage requirements are processed (validated) by the CCH by placing assignments into tables associated with the job. When the loading is complete, the Scheduling Processor builds an entry in the PPLT and enters pertinent data into the PPT. The LINKTB is then scanned for additional jobs to load.

The scheduling processor can choose to swap a resident job if the job is sufficiently low in priority or waiting for an operator response or facility request. The job and its volatile registers and mass storage assignments are written onto the disc and the core area is made available for another job. When the job attains sufficient priority, the necessary operator response is received, or the facility is assigned, the swapped job is eligible for reloading.

Program swapping is performed on the basis of the computer's memory organization. Memory addresses above 2100^8 are interlaced according to a program modification register which is added to the program address counter as execution proceeds. The executive interlaces programs into every eighth memory location as depicted in Figure 3. Additionally, the memory is constructed from banks of 4096 words. These two organization structures interact to form multiple pages per interlace position with program protection enforced between interlace positions.

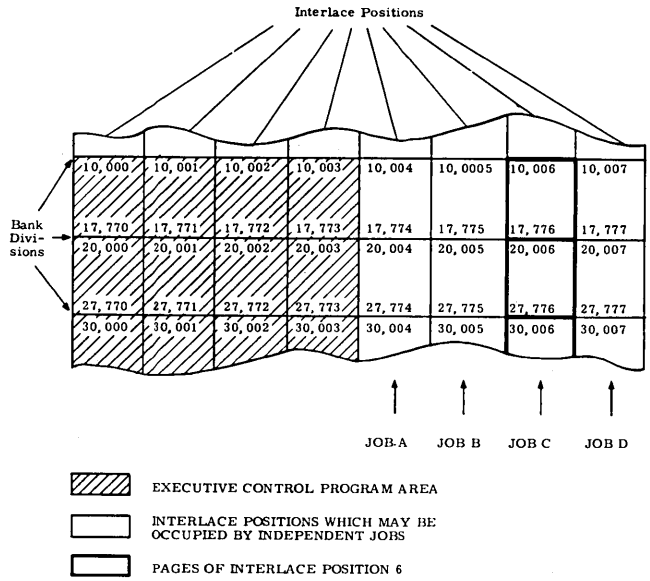


Figure 3—Interlace divisions

Swapping of programs considers memory requirements according to the following algorithm:

1) A job requiring x (x = 1, 2, 3, 4) interlaces will first swap a resident job using exactly x interlaces.
2) If x was 1 and there were no swappable jobs resident with only one interlace position, a try at x equal 2 is made. If no job of 2 interlace positions is found, the job is not considered further for scheduling on this round.
3) If x was 2, 3, or 4, and no job was found with matching interlaces, an attempt is made to acquire the required number of interlaces by swapping single interlace jobs.

Mass storage allocation and addressing

All programs, including the executive, address the disc file symbolically via the disc file handler program. Symbolic addressing in this system is implemented in two phases, Alphanumeric names coded by the programmer are converted to a numeric representation by the language processors. These "name numbers" are then translated by the executive (partially

at load time and partially at program execution time) to access the disc file. Four categories of data areas are provided; 1) permanent system areas, 2) temporary program areas, 3) console assigned areas, and 4) application data base area.

All areas, excepting the data base area, are 4096 words in length addressable in 512 word blocks. Permanent, temporary, and console areas occupy one-third of the total disc file storage. The data base area consists of the remaining two-thirds and is also addressable in 512 word blocks.

Permanent system areas are established in a Data Reference Catalog by the site manager. Read, Write, or Read and Write access privileges are also specified for individual JOB programs. These areas remain assigned until released by the site manager. Temporary program areas are assigned to a program by the executive when the program executes. Temporary areas are assigned when a program is loaded (based upon stated size requirements) or as a result of dynamic program requests to the executive. Similarly, the areas are released upon program termination or in response to a program request. Console areas are assigned and released by the executive at the request of an operator. Any programs requested from a particular console have access to console areas assigned to that console. The data base area is permanently allocated to a physically contiguous area on the disc file.

All requests to access the disc file are programmed through the use of two I/O statements as declared through the use of two declarative statements. For example, the statements to move temporary data to a permanent area are as follows:

```
[OPERATOR] ● [OPERAND 1]●●●
      [OPERAND N]
PERM ● BOX ● BOX 1
                                  declaratives
TEMP ● LOT ● LOT 2
                  Block No.
GET ● LOT (7) ● BUFFER
                                  I/O calls
PUT ● BUFFER ● BOX (0)
```

Three significant aspects of this method are: 1) lengthy request parameter packets carrying unwieldy alphanumeric names are not necessary; 2) request validation and processing by the executive is expedient; and 3) data can be physically relocated on the disc file without any changes (or regeneration) to the using programs. The first two aspects are made possible through an indexing scheme employing the name-number. The relocation capability is made possible because the disc file is "mapped" in 4096 segments. Each segment (area) is fixed to a physical (absolute) disc address and are mutually independent.

The price paid for this flexibility is the fixed 4096 word length of areas and the responsibility for proper coordination and control delegated to the site manager. However, these considerations may, as in this application, be worth the price where data standardization is possible and strong control by a human is desirable.

System catalogs

Several catalogs are employed by the executive to inventory mass storage allocations and object program library. Continuing the themes of program modularity and system integrity, a single program was designed to implement the catalog philosophies. The Centralized Catalog Handler (CCH) has exclusive control over the system catalogs and complete responsibility for their creation and maintenance. The CCH provides the following services:

1) Supplies the program loader with information stating the whereabouts and size of a program to load
2) Supplies information to the source language librarian and language processors stating the whereabouts of a program's source language
3) Supplies information to the source language librarian and language processors stating where to store a program's source language or generated language
4) Catalogs the location of program source language, generated language, permanent data, and console data.
5) Validates access to permanent data and console data and provides linkage for proper access
6) Obtains temporary storage and provides linkage for access
7) Ensures the integrity of the system disc file storage through proper maintenance of the catalogs and tables.

Figure 4 illustrates catalog formats.

Peripheral and core allocation

Allocation of peripherals in this system concerns only three devices; two magnetic tape units and one printer. It is sufficient to summarize the functional provisions associated with the allocation process as follows:

1) To avoid waste of time in loading a program, only to find that during execution the required peripheral is inoperable, facility requirements can be stated at the time an operational sequence is constructed. At the time the sequence

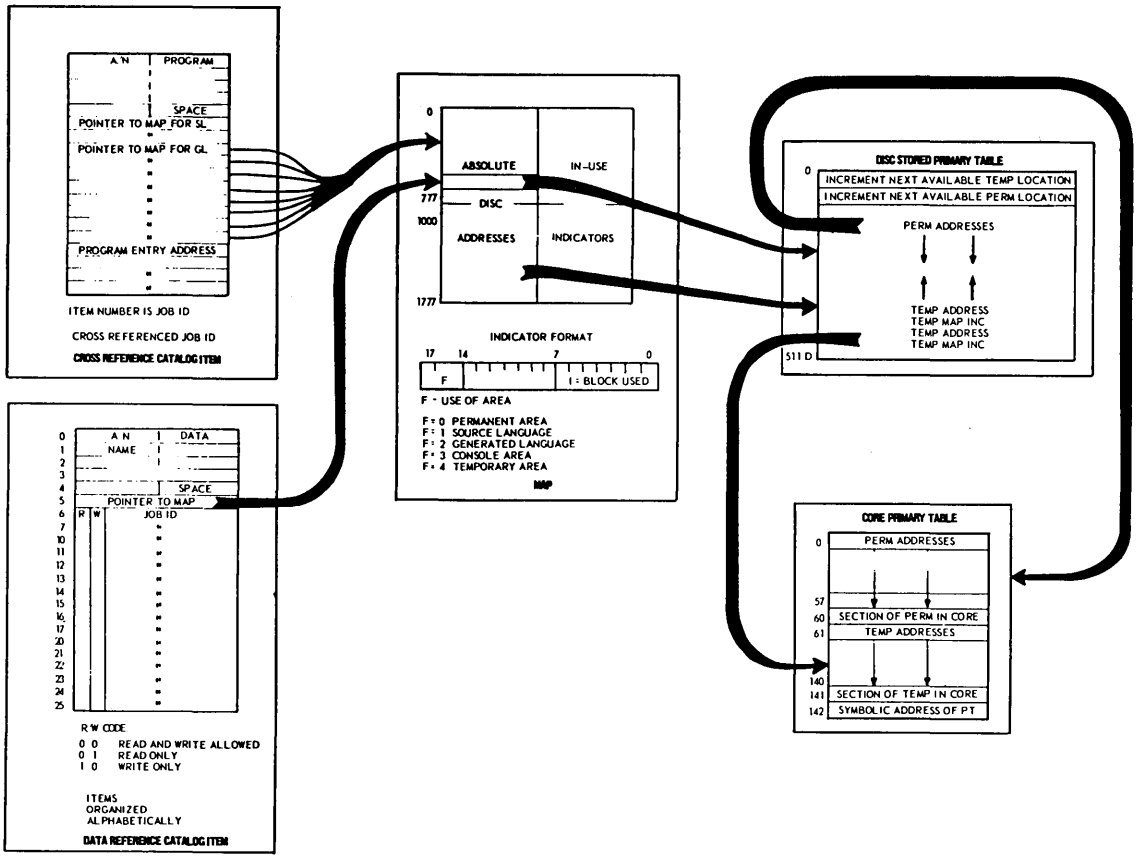


Figure 4—Symbolic addressing catalogs

is requested the executive will discover the facility's inoperable status and inform the operator. Programs can be designed to divert data to the disc file for later dumping by system utility routines, however the executive does not automatically perform this function. The operating status of peripherals is designated by the site manager through use of a supervisory function at his console.

2) Requests to obtain or release peripherals are submitted under program execution. Any JOB in a sequence can submit these requests. Request can specify any combination of the three assignable peripherals.
3) Facilities are optionally assignable for the requesting JOB program only, or for the remainder of the Operational Sequence. If not released by program request, the executive will automatically release them upon JOB or sequence termination as applicable.

Core memory allocation is largely a function of computer memory organization. When an operational sequence is constructed, program size requirements are specified in terms of interlace positions (seven 512 word pages per interlace position) for each JOB. Since memory/program protection is enforced between interlace positions, suballocations to different programs are not permitted. JOB programs may occupy a maximum of four interlace positions of resident program. Program segments may be loaded from the object library by call to the executive. Also, for extremely large programs, a special interpretive mode of operation can be selected at compile time. This mode of operation provides for automatic "ping-ponging" of program and data pages between core memory and disc storage as required during execution.

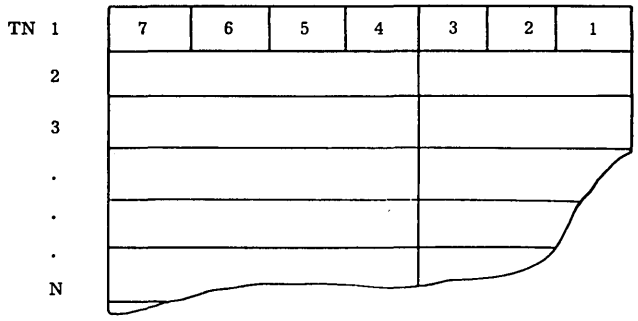
The executive itself employs a versatile method for page allocation which allows for "shuffling" of pages during debugging and facilitates executive overlays. The method is essentially a table of the executive tasks and a map of the memory divided into pages. The following information is recorded about each executive task:

1) resident or transient indicator

2) busy or not busy indicator
3) entry point
4) interlace position
5) page position within interlace
6) size
7) symbolic address of task on disc file

This information is indexed by task number. Additionally, for each page of memory an occupied or unoccupied indicator is maintained.

Executive Task Parameter Table



Note: Table construction is parallel (horizontal); numbers refer to preceding discussion points

SUMMARY

In reviewing this effort, one can identify a few major design features which greatly simplified development. Such things as constructing modules to both process and produce tabular data considerably aided the debugging process. This was largely due to the resulting module independence which made it possible to check out isolated modules. Table designs play very important roles in at least two ways; the obvious concern of all designers about the effort required to manipulate frequently accessed data, and the grouping of data according to the way the system flows. The latter means that modules manipulate a minimum number of tables. Also, at any point in the cycling of the system it is possible to obtain pertinent data on any given situation with a minimum of effort. Illustrating this point, the Processing Priority Table proved invaluable in assessing the status of the system at the time of a malfunction (since the system had almost invariably cycled "far down the road"). Several decisions in the processing of input/output simplified the system greatly. These include first-in, first-out queuing, minimal automatic recovery, and prohibiting swapping of programs with active input/output.

Since systems vary greatly in equipment types and capabilities and also in application, designs which may be optimum in one system may not be (and usually aren't) good for another system. Many trade-offs were made in designing this specific system. The areas of decision would have to be reanalyzed if this system actively used magnetic tape or card equipment or utilized different consoles. Another major design factor was the amount of human influence desired on the system operation. In this system, the design appears satisfactory-based upon observance of limited operation. The programming required to implement the application is not yet complete; therefore, data on sustained operation in executing actual application problems is not available. However, in its present stage the system has proved to be effective and flexible for developing programs on-line. Consoles can be rapidly changed between a program development mode and a problem solving mode. Total system integrity is ensured as governed by the authority vested in the site manager. Variations in the approach to solving a problem can be introduced, tested, and approved, all on-line, by means of the operational sequence technique of scheduling. Simple, relatively non-exotic methods were employed in the interests of achieving an implementation deadline.

ACKNOWLEDGMENTS

This work was sponsored by the Air Force (Rome Air Development Center) as part of a research and development contract with Univac. Specifically, Messrs. L. Odell and J. McLean contributed significantly to the system design.

Mass storage revisited

by ALBERT S. HOAGLAND
IBM Watson Research Center
Yorktown Heights, New York

INTRODUCTION

Mass Storage as a functional need in computer systems is continually increasing in importance with the growing trend to interactive terminal-oriented systems, serving as peripheral or external on-line memory for storing a systems data base and resident programming systems. The associated capacity, plus the ever expanding magnitude of such data, far exceeds the range where "electronic" memory is economically competitive. Included in the product category defined as mass storage are drum, disk, tape, card, strip, and chip recording structures. Direct access storage is becoming a standard feature of computer systems, with much the same type of distinctiveness as the CPU and main memory have achieved.

Five years ago I made a "tour d'horizon" of mass storage for the 50th Anniversary Issue of the Proceedings of the IRE.[1] It is of interest to preface this review by certain observations on the situation as seen then contrasted with now.

I felt then and still feel that magnetic recording will remain the technological base for mass storage for the foreseeable future. While this prediction has held true, the amount of exploratory activity in so-called beam addressable storage (optical and electron beam recording) has greatly increased. This current review (circa 1967) reflects a large relative increase in the material devoted to new technological approaches to mass storage.

Five years ago the highest storage density in a commercial file was 25,000 bits per square inch, while today this figure is approximately 250,000 bits per square inch—an order of magnitude increase (giving two orders of magnitude improvement over the last decade). However, at that time I felt eventually magnetic recording storage densities in excess of 10^6 bits per square inch would be realized and will restate the same conviction, reinforced by the advances already achieved.

An unanticipated factor which emerged to play a major role in the accelerating importance of development activity in mass storage is the challenge posed by programming systems residence. The complexity and size of operating systems is a fact of life that was little appreciated in 1962. The storage needs here serve to emphasize fixed head files with their capacity-access trade-off favoring short access time.

The success of the replaceable disk pack file has been remarkable and placed this device in such a predominant position that the long range role of strip-type storage structures has not emerged nearly as clearly as was expected.

In the last decade significant advances have also been made in tape drive performance as well as in the elegance of their design for reliability and serviceability. However, we are concerned here with the "image" of mass storage and these product advances do not essentially change the perspective of tape devices in an overall sense.

Overall, in the last five years, technology appears to have advanced as fast as suggested while the systems organization and use of mass storage hierarchies retains almost as much fertile ground for sophisticated design and application as appeared then.

Functional considerations

Computers that manipulate data are primarily limited by the number and size of files that can be made readily accessible for processing. Further, mass storage devices are now called on to fulfill important systems functions. Random (or direct) access units are used in compiling and assembling programs where their ability to reach large directories and subroutines rapidly is necessary for responsive program-preparation and execution. For time sharing, programs and data of many users can be stored on-line with a mass storage, to be run as requested in accord with some "optimum" allocation of facilities.

Capacity, access time, "latency," data transfer rate, and cost per bit are the basic performance characteristics of mass storage devices. Each mass storage

unit has its own particular attributes, and many applications require that several different devices, or a hierarchy of devices, be connected within the same computer system. The short access times and high data rates of fixed head drums or disks save valuable processor and internal memory time when it is necessary to swap programs, as in time-sharing. However, the higher cost per bit of these devices generally makes them too expensive for file storage, and other structures of higher capacity (and longer access time) but lower cost per bit are used. Cost of storage (in superficial terms) in inversely related to the number of bits stored per independent read/write transducer.

Systems-derived performance factors such as throughput depend not only on the mass storage specifications but also on indexing procedures, memory allocation and chaining provisions, file activity (ratio of records actually processed to total), provisions for queuing and ordering of access requests, checking techniques, etc. Thus, the associated control logic is an integral facet of any mass storage subsystem.

Technological considerations

The continuing need for high capacity storage has required the intensive exploitation of *recording* technologies for the economic implementation of mass storage. Thus, storage units involve the physical integration of recording media, transducers, precision mechanics, servo systems, and electronic encoding and decoding techniques to achieve a meaningful set of capacity access time tradeoffs.

Access to any data location is provided by relative motion between the storage surface and an associated transducer able to record signals on and sense the state of the storage medium. A single transducer may service many data "tracks" by a positioning mechanism operating normal to the direction of scanning. The recording density (bits per square inch) is principally a function of the registration tolerances (three-dimensional) that can be realized between the storage film and the coupling transducer. The access time variability to memory locations arising from the requisite motion necessary to scan large areas, makes the data organization of a mass store a key factor to effective systems utilization.

The greater the bit storage density (and hence the number of bits associated with a given read/write transducer) the lower the cost/bit and, correspondingly, the shorter the average access time for any given capacity. The average random-access time will range from milliseconds to seconds because mechanical motion is required for accessing masses of data, i.e., the larger the memory capacity, the greater the required surface area that must be accessible to a read/write station.

Although it is only in recent years that magnetic recording has come into wide general use, its invention by the Danish engineer, Valdemar Poulsen, dates back to 1898. The paramount functional advantage of magnetic recording surfaces is their unlimited reusability. This property permits the direct modification of stored information. Additional advantages of magnetic recording for mass storage of data over other potential storage film media are: the simplicity of recording transducer (a magnetic head); the flexibility in mechanical structure possible (and hence, choice of performance specifications) due to the ability to place the storage film on almost any supporting surface in conjunction with ease of mounting a magnetic head; the high bit storage densities and read-write transfer rates obtainable with magnetic recording; and great ruggedness with respect to handling and environmental conditions. The further features of replaceability and off-line shelf storage (e.g., tape reels and disk packs) make this mass storage technology extremely attractive and very economical.

Magnetic recording represents the integration of several basic engineering fields and has been generally characterized by rapid progress achieved by evolutionary advances rather than dramatic innovations. The one "breakthrough" that can be identified with the computer field is the "air-floated" head. Otherwise, advances in the magnetic recording art have largely emanated from increasingly higher precision and quality in components.

Historical growth and present status

The original work which ushered in mass data storage was firmly under way by 1947. This activity was associated and concurrent with the explosive "take-off" of the digital computer field at that time. Early work was oriented to the needs of the scientific computer market. The principal mass memory device was the magnetic tape unit, to provide both an auxiliary "back-up" storage for main memory and terminal buffering (data rate "matching" between input/output equipment and the central processor) in large-scale scientific systems. The later emergence of commercial data processing brought with it a wider variety of functional usages and mass storage hardware. Magnetic drum memory development for small and medium speed processors served to significantly add to the technological base of digital magnetic recording.

Commercial or business data processing, as it was evolving as a main facet of activity in the electronic

computer field in the early 1950's, gave a tremendous impetus to mass storage development and had a major impact on its direction. File storage for records maintenance was the central requirement.

Magnetic tape was first exploited for mass records storage. The only practical way to use tape is to address by record content. In updating a file, for example, the master and transaction tape reels are serially read (information is arranged and maintained in ordered sequence) and a "new" tape is created, on another transport, with the unmodified as well as the altered records being transferred. It was easy to make insertions and deletions in this process as well as to handle variable length records, as physical sections of tape have no specific identify; although to modify or insert a single record, the entire tape must be rewritten. A tape reel is relatively cheap and therefore low-cost, off-line, archival storage is attractive. For low file activity, tape devices are very inefficient, due to the constraint of sequential access. Further, effective file inquiry operations are not possible.

The character of much business data processing indicated the need for an entirely different type of mass storage. The desirability of storing large volumes of information with any record available rapidly gave stimulus to the development of a mass random access memory.

The air bearing supported head (using an air cushion to control head-to-surface spacing) was the innovation which, associated with the above memory concept, brought about this entirely new type of mass storage. An air-bearing head can follow considerable surface fluctuation—up to 100 times the spacing. Since the readback amplitude wavelength dependence on separation is given by $e^{-2\pi d/\lambda}$ (where d = separation and λ = wavelength) high recording densities would be impractical without such a method of maintaining close and accurate spacing. By this air-bearing spacing technique, it was possible to develop a high-capacity rotating disk array since a head could closely follow the appreciable runout of large disks.

The first version (the RAMAC, announced in 1956) could store five million characters with an access time to any record of less than a second, having one head mechanism servicing the entire disk array. Secondary technical features of note were the use of self-clocking and a wide erase narrow read-write head unit. These design approaches, combined with the use of an air-supported head, provided techniques that compensate for the head-to-track registration tolerances of such a gross mechanical structure, and thus permitted the high track density and high bit density necessary for large capacity.

Random access memory involves addressing by physical location to a single record or a particular block of records (one track), which is then scanned. Any record can be read, written or modified without affecting any other record. The "set of keys" (record identifiers) of a file will, in general, bear no direct relation to the closed set of machine addresses. Various randomizing techniques (key transformations) are used to convert scattered keys covering an extensive range to a dense and relatively uniform distribution of numbers to obtain automatic addressing capabilities.

Initially pressurized air was fed into the spacing gap to maintain separation. Around 1960 a significant advance was achieved as self-lubricating air bearings came into general use on both disks and drums, bringing great simplicity and cost advantages, and in particular making head-per-surface disk arrays feasible.

Many approaches to chip, or strip, or card type mass storage devices have been undertaken over the last decade and NCR has been the leader with their CRAM.[2,3] However, the simplicity of rotational motion over card shuffling has favored disks, and no strip-type storage structure has emerged as a clearly identifiable industry symbol. Indications are that such devices must be complementary to disk storage and it is still premature to detail such configurations in the context of their evolutionary significance.

The next major innovation in mass storage (1962-63) was the replaceable disk pack concept, which emerged as a practical alternative with the rapid progress achieved in storage density—from 2000 bits per square inch (1956) to 50,000 bits per square inch (1962), permitting adequate capacity to be stored on a few small disks.[4]

Three discernible themes have emerged in the area of mass storage.

1. The replaceable disk pack file is clearly appearing as the principal mass storage structure characterizing third generating computer systems. The disk pack file appears well on its way to becoming as ubiquitous as were tape drives on the computer systems of the late 1950's. With the major trend toward high performance disk files, there is concurrently much less forward momentum in magnetic "strip" storage as a competitive alternative, contrary to some earlier projections.

2. A second aspect is the vastly increased discussion and presentation of advanced work in new technologies that may become significant for computer storage.

3. Fixed head files, either drum or disk, have entered a stage of greatly renewed interest due to the demands now being generated for high capacity storage with fast access to store extremely large and

sophisticated programming systems as well as meet the data and program swapping needs of time-sharing.

The disk pack cluster demonstrated another answer to very large on-line storage and could become a principal approach to serving requirements generally met today by large disk array files. For example, the IBM 2314 consists of eight disk packs (plus one spare) each with a completely independent access mechanism, all in one assembly. With this concept, by overlapping accesses, the mean access time can go down with increasing capacity at a relatively fixed cost per bit. The performance region of several hundred million bytes of data with access in milliseconds is a key to many on-line "data bank" type information systems and the appeal of such facilities spurs the rapid trend to direct access file-oriented systems.

The "state of the art" magnetic recording density for production disk files (1966) moved to the region of 250,000 bits per square inch (2500 bpi by 100 tpi), a factor of 125 greater than the density of the first commercial disk file announced only a decade ago. Further, major advances in density and thereby capacity are clearly indicated based on many current laboratory investigations in the range of 10,000 bits per inch. Techniques that may make equivalent strides in reducing access time are still undefined. The trend to large on-line data banks will undoubtedly accentuate reliability considerations far beyond traditional experience—a question with electromechanical devices of constantly growing concern.

New beam addressable concepts for storage are also being described periodically that may be promising for the future. Numerous groups revealed activities looking towards developing reversible magneto-optic storage techniques. Writing is accomplished by thermal heating (e.g., with a laser beam) in the presence of a magnetic field and reading by sensing the rotation (caused by the magnetic state of the film) of a polarized beam of light.

Among other mass storage approaches recently presented were: a trillion bit storage system for the Livermore AEC Laboratory based on electron beam recording on photographic film—with a flying spot scanner for reading; an electron beam multiaperture recording structure; several concepts using holography as a means for very high density optical data storage, a recording scheme which stores information in a plastic film with a laser by "drilling" holes (whose presence or absence can be optically sensed), as well as variants on these methods. These latter schemes do not allow direct updating of recorded information.

Most of these development activities are in a very preliminary stage and one would not anticipate commercial products that capitalize on such technologies to appear for several years. Nevertheless, these releases do dramatize a recognition of the growing importance of achieving higher performance storage products to meet the growing system demands posed by the trend towards on-line multiuser systems oriented to remote terminals.

Future trends

Each user wants all data immediately available to him. The appetite for more mass storage capacity with high speed access appears as insatiable as the appetite for more computer speed.

A major problem in file-centered systems is determining the proper balance of mass storage devices. A hierarchy of storage modules (of varied performance parameters) closely integrated can best optimize overall systems performance, recognizing that cost trends to be inversely related to access time for a given capacity. We need to better understand the flow of data and develop design guidelines here, recognizing that the access "gap" from electronic memory through electromechanical storage devices will cover a relative access speed differential of 10^3 to 10^5. Mass storage control logic is now beginning to extend to the functional ability to automatically arrange records within the hierarchy of memory/storage according to their activity, significantly improving access utilization. Much sophisticated systems development work is still necessary if we are to fully exploit our technology capability.

A major advance in hardware reliability is urgently needed if the full potential of mass storage devices is to be realized. Capacity-access time improvements must be at the price of higher reliability because of the nature of on-line systems.

In this regard the mechanisms of strip-type mass storage devices are considerably more complex than those of disk files or drums (the proper surface must be selected at random from many hundreds, accelerated, guided repeatedly past a read/write station, decelerated, and returned to the file), and there is consequently a higher probability of device malfunction.

Beam addressable storage technology

A read-only photographic rotating disk memory was developed several years ago to serve as a dictionary for langauge translation.[5] The model stored 30 million bits (approximately 1500 bpi × 1500 tpi) with an access time of about 50 milliseconds. Optical sensing using a CRT flying spot scanner permitted "servo tracking" with extreme sensitivity, allowing a much

closer balance to be obtained between bit and track densities than in magnetic recording.

A number of new electron beam and optical techniques have since been proposed which eliminate the need for mechanical motion for access, at least within a large block of information. These schemes have an appeal since beam addressable techniques have the unique capability for microsecond access to densely recorded information within the field of view of the beam. An electron beam can readily be deflected and modulated. A laser provides a light source whose output can be focused to a small size at high power densities. There is confidence that means for modulating and deflecting laser beams will be found that are practical.

Electron beam spot sizes less than one micron in diameter have permitted writing (in a vacuum) at a density of several million bits per square inch (approximately 2000 bits/in. by 2000 tracks/in.) on silver film. Readback must be done optically at a separate station after film processing. Future improvements may give even smaller beam sizes over a wider field of view. However, as in magnetic recording, we can now write information at much higher densities than we can effectively readback because we encounter still unconquered signal detection problems.

Optical techniques offer greater possibilities for high speed *parallel* data flow as well as the prospect of "distributed" bit storage (e.g., by holography) where the effects of media defects and registration misalignment at high densities may be minimized. A hologram is created from the bit pattern and recorded. The original image is reconstructed for readout by photodetectors. The greater possibilities of developing suitable "reversible" optical media (e.g., magneto-optical films) provide a further incentive for pursuing exploratory studies in optical recording.

Interest in photographic film stems exclusively from the potential it offers for high storage density. Photographic emulsions, which can be written with either light or high-energy electrons, are capable of resolutions of the order of 100 million spots per square inch. High speed recording is possible at a reasonable power level. But silver film is not reusable and a long delay occurs between the writing of information and its availability for write verify or reading. A significant capacity/cost advantage must be achieved in a read-only type store to justify the acceptance of the systems performance limitations on updating, posting, and immediate write-checking.

Organic photochromic materials fatigue with use and are relatively slow. Thermoplastic type media (which can be recorded upon by an electron beam) while reversible, involve a distinct developing phase, and the "deformation" storage mechanism limits resolution.

There are severe problems in locating and tracking information stored at very high density. Servo-techniques (also being pursued for higher track density in magnetic recording) based upon track-seeking principles are essential for beam scanning approaches.

Magnetic recording technology

Magnetic head design skills have advanced to a point where the principal concerns are to evolve batch fabrication techniques that will provide precision assemblies at a much lower cost. Magnetic heads can yield in excess of 10,000 cycles per inch resolution with frequency bandwidths extending considerably beyond ten megacycles.

No physical phenomenon for magnetic recording appears competitive with the simplicity and flexibility of the conventional magnetic head. Rather, the next major state of progress in the transducer area will be automated head fabrication (rather than watchmaker-like assembly), resulting in dramatic cost reductions. Such a breakthrough would have an impact upon the hardware composition of mass storage mechanisms, by making it economical to radically increase the overall ratio of heads to tracks.

The mainstream effort in magnetic surface work is to achieve higher quality, thinner recording surfaces. (Magnetic films in use today are oxide coatings formed from a dispersion of either Fe_3O_4 or $\gamma - Fe_2O_3$ in an organic binder, and Co-Ni platings.) Thinner magnetic recording films are indispensable for higher resolution whenever "bulk" erasure is not feasible and in any event must be correspondingly higher in quality (uniformity, finish, etc).

Reduced head-to-surface spacings (from approximately 100 μinches today down to approximately 25 μinches) will bring about a further large increase in "noncontact" bit density. Track seeking and following servo access techniques, which can circumvent the track density limitations imposed by the build-up of cascaded mechanical dimension tolerances, will be applied to effect significant increases in track density. Thus, recording mechanics still offer considerable room for further advances in storage density.

Improvements in the speed of head positioning actuators appear possible. However, progress in this area is not rapid and even a factor of two in speed will be difficult to achieve. Rather, if proposed batch fabrication processes can be successfully applied to achieve a drastic reduction in the cost of magnetic heads, and LSI (large-scale integration) enables a

corresponding decrease in the cost of head switching and read/write electronics, there will be a substantial increase in heads per file module, minimizing the access dependence on head positioning.

Recording methodology is as yet an inadequately explored means to further upgrade storage density. Since the advent of digital magnetic recording for mass data storage, the primary avenue taken to increase density and thus performance has been through improved recording mechanics. This work emphasizes improving the "resolution" of a recording system and is approaching basic physical limitations. A relatively untapped potential exists for increasing magnetic recording density by the application of more sophisticated communication concepts to the encoding and decoding of the information to be stored.

Recent communications work exploiting adaptive equalization techniques and advanced coding theory has successfully demonstrated dramatic increases in data rates on transmission lines. In a similar vein the use of advanced communications techniques for the magnetic recording "channel" may yield significant gains in bit density relative to pulse width.

The calculated density limit based on a typical pulse response with standard recording techniques under ideal conditions is extremely high. However (even on paper), a large reduction in this potential figure results from a simple consideration of variability between magnetic heads, tolerances on magnetic coating thickness, radial variations in spacing and speed (with disks), head-to-track registration stability, surface speed fluctuations, etc. In addition we must deal with "noise" sources, such as surface imperfections, etc. All these factors must be accounted for in current file systems, which generally are designed around a single read/write channel serving a multiplicity of positionable transducers. Thus the actual operational density falls far short of that projected based only on a typical pulse waveform.

If adaptive techniques and advanced coding methods permit us to compensate for a number of these factors and operate anywhere near the "theoretical" limit of an idealized magnetic recording channel, the gain (through electronic rather than "mechanical" means) realized could be very significant.

Despite the bright projections into the near future for magnetic recording (the next generation of magnetic recording data storage devices may consolidate around 5000 bits/in. and 200 tracks/in.) the outlook for this recording process in the next ten years is not nearly so reassuring because it does not seem to lead as far as we would like to go.

At the present time there is no approach to mass data storage that could obviate the use of physical motion, using a continuous recording medium and transducer. While a completely electronic mass data store is an obvious goal, such memories seem possible only on the low capacity, high-speed side of mass storage (now filled by drums and disks with fixed heads).

There are substantial reasons to believe magnetic recording will undergo another decade of progress comparable to the past one, although there is something in man that rebels against the mechanical shuffling of data to recall information. We want instant memory and we desire this capability in the computers that we use.

For the foreseeable future, however, it is clear that mass storage based on magnetic recording has a vital and increasingly important role in information processing.

REFERENCES

1 A S HOAGLAND
Mass storage
Proc IRE 50 1087 1962

2 A F SHUGART YU TONG
IBM 2321 data cell drive
Proc SJCC Spartan Books Washington D C vol 28 p 33 1966

3 L BLOOM I PARDO W KENTING E MAYNE
Card random access memory (CRAM): functions and use
Proc EJCC Washington D C 1961 p 147

4 J D CAROTHERS R K BRUNNER J L DAWSON M O HALFHILL R E KUBEC
A new high density recording system: the IBM 1311 disk storage drive with interchangeable disk packs
Proc FJCC Spartan Books Las Vegas Nevada 1963 p 327

5 G W KING G W BROWN L N RIDENOUR
Photographic techniques for information storage
Proc IRE 41 1421 1953

BIBLIOGRAPHY

T H BONN
Mass storage: a broad review
Proc IEEE 54 1861 1966

W W CARVER
Comparing storage methods
Electronic Industries 21 120 1962

J S CRAVEN
A review of electromechanical mass storage
Datamation 12 22 1966

L C HOBBS
Review and survey of mass memories
Proc FJCC Spartan Books Washington D C p 195 1963

W W PETERSON
Addressing for random-access storage
IBM J Res and Dev 1 130 1957

C B POLAND
Advanced concepts of utilization of mass storage
Proc IFIP Spartan Books Washington D C p 249 1965

High-speed thermal printing

by RICHARD D. JOYCE
National Cash Register Company
Dayton, Ohio

and

STANLEY HOMA, JR.
U. S. Army Electronics Command
Fort Monmouth, New Jersey

INTRODUCTION

Thermal Printing is a unique new concept in non-impact printing, whereby electrical signals are directly converted to heat to produce a printed output. This paper covers a program to build an advanced development model of a High-Speed Thermal Teleprinter, utilizing the Thermal Printing technique, for the U. S. Army Electronics Command. The teleprinter, which prints at 240 characters per second, was the result of the development.

History

The concept of Thermal Printing was developed to meet a requirement for a low-cost, quiet, high-speed computer print out device. At the time that the contract was awarded to NCR (June 1964), however, the most advanced Thermal Printer was a keyboard operated, strip printer, which had a maximum operating speed of 15 characters per second, and made only a single copy.

Viewed against this background, a 2 year program to develop a Teleprinter which printed across a full page at 60, 120, and 240 characters per second and made from three to six copies certainly appeared to be a very energetic undertaking. However, from the Government's point of view, the features of thermal printing made the potential gains commensurate with the risks.

Features of thermal printing

In addition to being non-impact, Thermal Printing has the following salient features which make it attractive for both military and industrial applications:

- **Reliability**—Inherent high reliability due to the absence of moving parts.
- **Low RFI**—The absence of high voltages and solenoids allows even the rigid requirements of FED-STD-222 to be met.
- **Silent**—The printing process itself is completely quiet; the only sound is that of moving the paper.
- **Low power**—Due to high efficiency of direct electrical to thermal energy conversion.
- **Speed**—Absence of moving parts and their associated inertias allows high speed operation.
- **Clean**—No inks, powders, or fixing actions are required.
- **Multiple copies**—Most non-impact printers do not have this capability.

Thermal technology

To achieve Thermal Printing, two novel components are utilized. The first, a print head, is an array of resistors selectively heated electrically to generate a thermal image. The second is a thermal sensitive paper which is held in contact with the print head to provide a printed record of the thermal image.

Numerous approaches have been tried in fabricating print heads, and several promising new approaches[1] are still being developed. However, the print head configuration described herein has been used in all Thermal Printers built to date.

Each individual printing element consists of a tin oxide resistor deposited on the edge of a ceramic substrate (Figure 1a). Electrical connection to the printing element is made through the copper leads on the top and bottom surfaces of the substrate. Fifty print elements are fabricated on a single substrate to form a

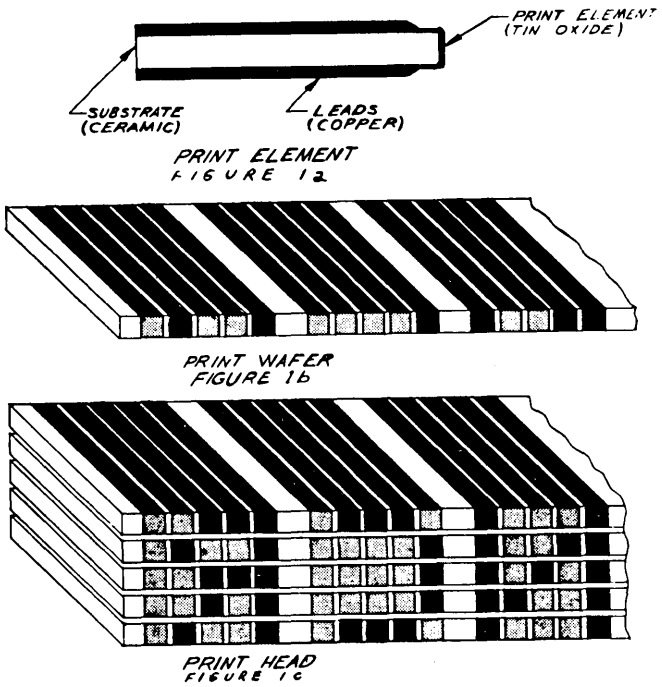


Figure 1a—Print element
Figure 1b—Print wafer
Figure 1c—Print head

print wafer (Figure 1b). They are arranged in 10 groups of 5 with a space in between each group.

Five wafers are mounted together to form a print head (Figure 1c). Each print head contains ten 5 × 5 matrices; each matrix is capable of printing one character. Eight print heads are mounted side by side in the printer to achieve the 80 character line.

By applying power to combinations of print elements in a matrix, the thermal image of the desired character can be formed in the appropriate position on the print line. The Thermal Printer has the capability of printing any one of 64 alphanumeric characters or symbols in any position on the print line.

A 5×5 matrix was chosen over the more widely used 5×7 matrix for the following reasons:

1. Choosing the 5×7 matrix font would increase the number of print head wafers and driver circuits by exactly 40% plus increase the complexity of the selection logic by approximately 40%.
2. The intelligibility of the printed characters employing the 5×5 font rather than the 5×7 font was not significantly reduced.

High-speed thermal teleprinter

The High-Speed Thermal Teleprinter was developed to print at speeds of 60, 120, and 240 characters per second, print 64 Alphanumeric Characters using the serial input mode, provide a MIL-STD-188B interface, and produce multiple copies.

Let us now discuss what problems were encountered along the way, what significant advances were made, and how this printer operates.

When this thermal printing contract was initiated, there were some major areas where either ideas had not been proven or no firm ideas existed at all. Major problem areas were known to be the print wafers and a multi-copy process.

Print wafer problems

There were four basic problems in the print wafers alone. They were abrasions, interconnections, alignment, and burnout.

The abrasion problem was due to the print heads rubbing against the paper while printing. At the time of the contract award print head life was about 15,000,000 lines of print. Through tests of various resistor materials and overcoats, and continued plated mechanism development, the life of the print head was extended to 50,000,000 lines of print, which is equivalent to operations at the 240 characters per second speed with a 25% duty cycle for 2 years. The total printed output would be equivalent to that of an existing 10 characters per second, Standard A, Military Teleprinter operating at a 25% duty cycle for 48 years.

The next problem area was interconnections. For a full 80 character line, there are over 2,000 printing elements lined up in 5 rows. The print line is 0.1 inch high and 8 inches wide. This means that over 2,000 connections had to be made to the wafers through a cross sectional area of only 0.8 square inches. To do this a technique was developed to connect high density, copper clad, mylar cable conductors to the matching conductor pattern on the wafers. The conductors are on 0.028 inch centers—0.018 inch conductors with 0.010 inch spaces. There are several standard ways that the cables could be fastened to the print wafers. The connections could be bonded with a conductive epoxy, soldered, or welded. All of these processes, however, require very precise registration and would need to be performed in a clean room as foreign particles could short out adjacent conductors. In addition, the soldering and welding both damage the copper clad mylar cables, and the problems of registering a conductive epoxy pattern would be difficult. To overcome these problems, the process that was used for this development was first to apply a very thin coat of non-conductive epoxy on one side of the wafer, then lay the cable on the wafer and align the conductor patterns visually. Next, the cable and the wafer are clamped together

with a plastic clamp that permits visual verification of the alignment while the epoxy is drying. After the epoxy dries, we have not only a permanent, rugged, mechanical bond between the wafer and the cable but also a good electrical connection between each conductor on the wafer and its corresponding conductor on the cable. The mechanical bond is, of course, formed by the epoxy. The electrical connection is not so easily explained, since this epoxy is a good electrical insulator.

The key to understanding the electrical contact is that the conductor surfaces are not microscopically smooth but composed of tiny hills and valleys. When bonding is performed, the pressure forces the epoxy into the valleys and causes the hills to make contact. After the epoxy is dried and the clamp removed, the epoxy still holds the conductor surfaces in contact. Typical resistance is 0.1 ohm with a maximum resistance of 0.25 ohm. Connections of this type have been tested successfully at temperatures as high as 1000° F, at which point the mylar cable failed (not the electrical connection). Over 100 cables, having 30 conductors each, have been bonded to date, and only two defects have been found. This would indicate a 98% yield which is better than previous experience with a dip solder technique for this application. This process may find other uses. In general it could be considered whenever a permanent connection is desired between multiplicities of conductors and space is at a premium.

The next problem area is alignment. If all the printing elements are not located in the same plane thermal transfer to the paper will be poor, resulting in poor quality printing. Tests have shown that the allowable deviation from the printing plane is around ± 0.0002 inch. Since the print head plane is composed of five discrete wafers, the required alignment is not easily maintained. In previous work this had not been considered a major problem because only single character printers had been built. In these instances the area of the printing plane and the cost of the wafers were both relatively small. This led to hand tailored wafers and assembly procedures which could not easily be expanded to adequately meet our problem. Although maintaining the flatness of the printing surface on the wafer to these dimensions was no problem, the tolerance on the dimension which located the printing plane could not be held closer than ± 0.0005 inch.

Consequently, the procedure adopted was to measure and divide them into groups of five, such that each group was within the required tolerance. Each group of five would then be used in one print head assembly. The print head assemblies were then made so that they could be adjusted to overcome the alignment problem between adjacent print head assemblies. Although this approach has produced a functioning printer, it is felt that the problem has only been eluded rather than solved. However, there are other techniques currently under development for producing print heads that would not have an alignment problem.

This brings us to the remaining problem area—burnout, where the most significant contribution was made. When this program started, the state-of-the-art thermal print heads were satisfactory for 15 cycles per second operation, consisting of a 10 millisecond print pulse followed by a 57 millisecond cooling period. However, in this application, under worst case conditions, we had a 4 millisecond print pulse followed by 21 milliseconds of cooling time. The only easily measured parameters which were useful in this area of development are energy required per pulse to achieve printing, and energy required to cause failure (burnout) at a given duty cycle. In this design a 10 mws pulse of energy is used to print and, at the maximum duty cycle of 16.7%, 15 mws of energy per pulse would be required to cause an element to fail.

It is interesting to note that unlike most other printing systems, thermal printing does not require more energy to go faster. This is because the inertias associated with mechanical printers have no thermal counterparts. The term "thermal inertia" is a widely used misnomer which can lead to faulty conclusions about heat flow. Printing has been achieved with pulse times as short as 150 microseconds.

The limiting factor in high speed operation is not therefore the energy required, but the maximum temperature which the elements can withstand. With constant energy required, shorter pulse time causes the input power (rate of energy) to be increased causing the maximum temperature to increase.

Since a thermal print element is composed of layers of different materials in a three dimensional configuration, an accurate mathematical model is difficult to construct. However, a simplified mathematical model has been constructed from the equations for the temperature of a semi-infinite solid with a single heat pulse applied at the surface and parallel heat flow, perpendicular to the heated surface.[2] A number of assumptions (which are now known to be not precisely correct) must be made to apply to these equations to simplify them; the resulting curve approximates the shape of the measured Temperature versus Time curve for a printing element (Figure 2). The model consists of two equations, one for the time while the print element is heating and one for the time while it is cooling.

$$T_H = \frac{CP\sqrt{t}}{A} \qquad (1)$$

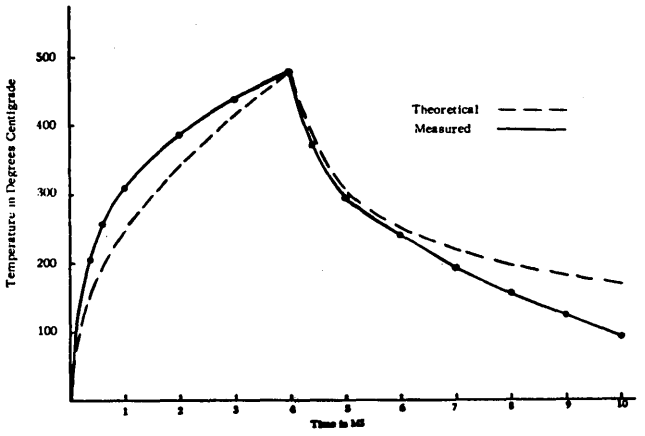


Figure 2—Thermal printing process

$$T_c = \frac{CP}{A}(\sqrt{t} - \sqrt{t - t_p}) \qquad (2)$$

where T_H = Temperature while heating
P = Input Power
A = Area of Printing Surface
t = Time
C = Empirical constant
T_c = Temperature while cooling
t_p = Time of printing pulse

The vertical tangent at $t = 0$, in both the theoretical and measured time-temperature curves, graphically emphasizes that the previously discussed "thermal inertia" does not exist. In can also be seen from equation (2) that the rate of cooling is a function of the rate of heating. This feature, which has been confirmed experimentally, means that for high-speed applications where a short heating pulse is applied, cooling also occurs more rapidly.

Since the energy required for thermal printing remains nearly constant regardless of pulse width, this application was more severe in two respects. First, applying the same energy in a shorter time causes the element to be driven to a higher maximum temperature. Second, since the applied energy is the same but the cooling period is shorter, the energy must be carried away faster. The graph, referenced in Figure 3, shows the performance of the elements on a print wafer that was available when this program started. The "y" axis represents energy and the "x" axis represents the print elements. For each print element there is a circle representing the print energy and a triangle indicating the burnout energy. Three important aspects of the print elements should be observed on this graph. The first is that the variation of both the print and

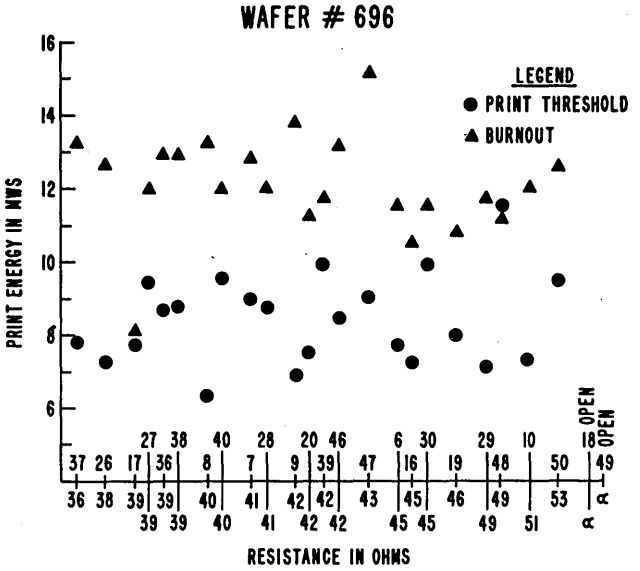


Figure 3—Print element temperatures vs time

the burnout energies are large. The second is that some of the print levels are higher than some burnout levels, indicating inadequate performance. The third is that some elements show adequate margin of safety between the print and burnout energies. Many changes in the print element were tried in an attempt to obtain a consistent, adequate margin of safety between the print and burnout energies. Finally a real breakthrough came. It was found that the thickness of the copper conductors on the wafers was a very important factor in obtaining high performance print elements. This makes sense since the copper conductors would be a major thermal conductor also. Using this knowledge it was also found that the dimension from the printing surface to the copper termination was also an important factor. By controlling and optimizing both these parameters it was possible to both reduce the variation in print and burnout energies and maintain a consistent high performance level. This is shown in Figure 4. Wafers having thse characteristics were used in the printer.

Although successful Thermal Printers have been produced, much work in the basic research areas needs to be done if the full potential of Thermal Printing is to be realized. Two of the more promising areas are:

(1) Refining the mathematical model to include the effects of paper and the copper leads; both of these are known to cause gross effects. Extend the model to handle the case where multiple heat pulses are applied.

(2) Testing thermal sensitive papers down to millisecond response time range to determine the limitations which the papers themselves may impose on the process.

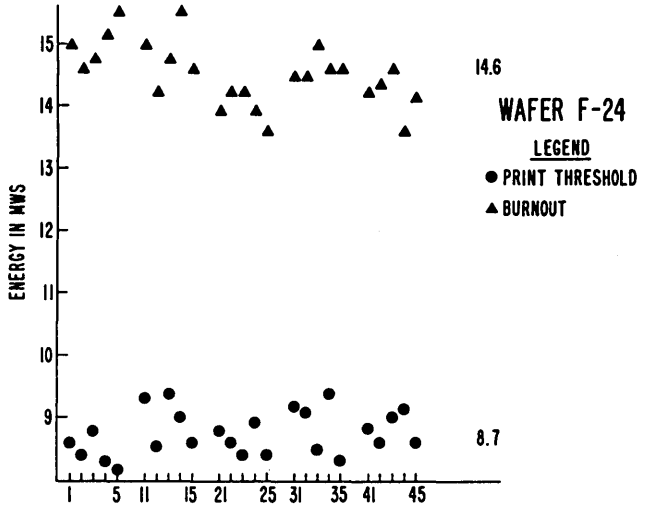


Figure 4—Early print wafer

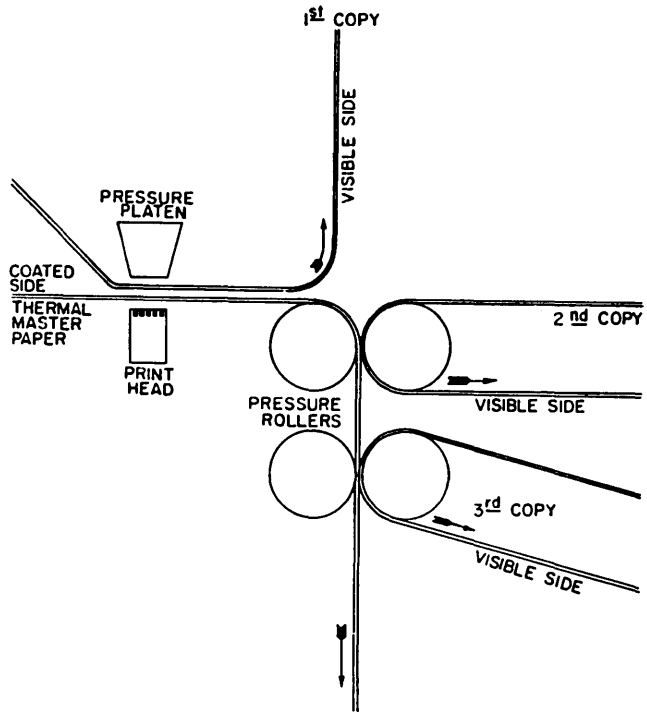


Figure 5—Improved print wafer

Multiple copy problem

The other major problem area was in making multiple copies. All previous thermal printers made only a single copy by contact with a heat-sensitive paper which changed color when and where it was heated. Therefore, a paper was used which had the particular property that it is sensitized by heat and makes copies by transfer under pressure. When the paper is heated by the print heads the back coating becomes tacky — not tacky to the touch but tackiness that will cause colored material to be transferred to plain paper when it is subjected to the fairly high pressures of the pressure platen and pressure rolls as shown in Figure 5. There are 300 to 400 pounds of pressure on these rolls in order to insure good copy. The tackiness remains for approximately 15 minutes so copies can be made during this time period. The paper must also retain enough of the colored material so that it will not all be released initially but can make a number of copies. A very desirable feature of this copy process is that only plain paper is needed to make copies thus reducing the operating cost. Three copies can be made in the printer and three more by a copy roll box which was also delivered with the printer.

Papers capable of producing up to 20 legible copies have been tested. Copies made in this manner will not smear with handling or fade due to time or temperature.

Although the single copy and multiple copy papers are functionally different, they can both be used on the High-Speed Thermal Teleprinter. Threshold temperature for both types can be altered to fit different applications. Normal threshold temperatures range from 125° F to 200° F.

Printer logic

This printer uses the thermal print heads to give a 5×5 matrix font in a full 80 character line arrangement as described previously.

Now that we have the print heads, let us discuss how the printer operates electronically to cause thermal printing. The block diagram shown in Figure 6 will aid us in this discussion. The printer receives the incoming data in serial form — a start bit, 7 bits in ASCII* format, a parity bit, and then a stop bit. This represents one character. In the Input Section, the digitizer converts the data to logic voltage levels and stores this information in parallel form in the receive register. It also checks the data bits to be sure that they are of odd parity and checks the stop bit to make sure that synchronization has been maintained over the character interval. After the stop pulse has been received, the contents of the receive register are loaded into the intermediate register to make room for the reception of another character. Next, the Control Section will shift the contents of the intermediate register into the print register. The contents of the print register are then decoded to determine the proper character to be printed. This character information is then sent

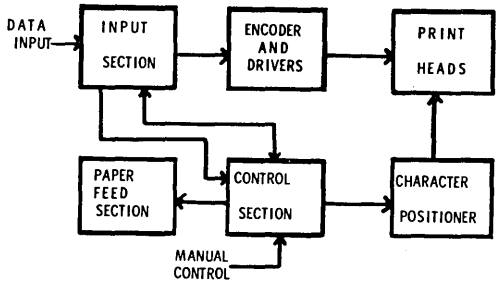


Figure 6—Printer block diagram

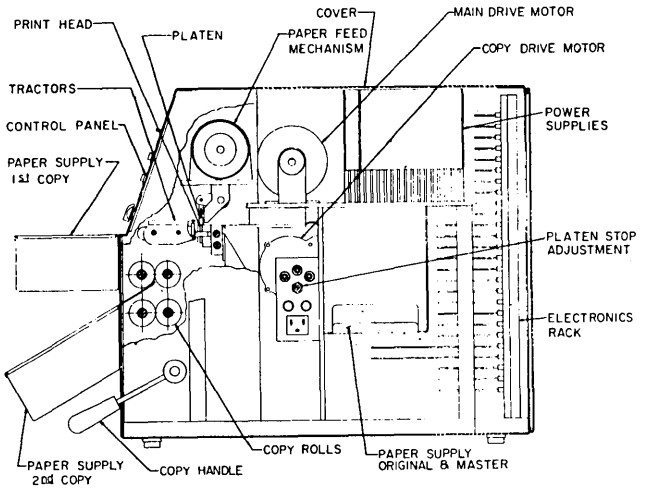


Figure 7—Printer layout

to the Encoder and Driver Section where encoding into the 5 × 5 font code is performed and the appropriate bit drivers are turned on so that the desired character will be printed. The position in which the character is printed is determined by the Character Positioner.

When the character has been printed, the character position counter in the Character Positioner will be upcounted. The outputs of the position counter are next fed into the character decoder which in turn selects the character driver which determines the position in which the character is to be printed. If the character received is a carriage return or line feed symbol, a line feed operation will be performed by cycling the clutch and brake circuit in the Paper Feed Section, and a carriage return operation will be performed by merely resetting the character position counter to zeros. Notice that no mechanical action is involved for a carriage return operation.

After several line feeds have been performed the copy paper tension switches in the Paper Feed Section will sense the slack in the paper and activate the copy roll gear motor which will drive the copy section.

Hardware

Now that we have discussed the operations of the printer, let us look at how the hardware for this printer fits together. A side view of the printer is shown in Figure 7. The location of the electronic circuits, power supplies, paper supply, paper feed mechanism, printing head, platen, copy section, and control panel is shown in this layout. It should be pointed out that all the logic in the machine is integrated circuits even though the design was started in 1964.

The way that the media is handled in this printer may copy roll gearmotor which will drive the copy section. paper are stored together in a 2-ply fan-folded format with the folds on 5½ inch perforations. This media is sprocket fed one line at a time. As this media comes under the print head the master paper is on the bottom. To effect printing the platen is operated towards the media in order to obtain a firm contact between the print head and the media. This platen remains operated until a line feed character is received, then the platen is moved away from the media and the media is stepped one line to prepare for the next line of print. As a message is printed, the original copy moves up the front of the printer and this copy can be viewed through a window. The master copy goes down into the copy section where two more copies are made. If only the original copy is desired, the master copy can be routed directly out of the machine after producing the original copy. Then, if desired, the master paper can be used off-line with a copy roll box to make at least five more copies. A photograph of this printer is shown in Figure 8.

Other developments in thermal printing

Let us now talk briefly about other developments in thermal printing and possible applications.

Contrary to what many people think, the speed of thermal printing is relatively fast. A 150 microsecond current pulse has been found to cause printing although this speed of printing has not been thoroughly tested to date. But even forgetting about this 150 usec rate and taking the 4 msec rate of printing used in the High-Speed Thermal Teleprinter, we can show that thermal printing compares in speed with high-speed line printers .The present High-Speed Thermal Teleprinter prints serially by character so with the addition of multiple electronic circuits, this thermal printing technique

Figure 8—High-speed thermal teleprinter

could be incorporated into a line printer configuration —printing each line of characters simultaneously. The speed calculation for such a printer is as follows:

Assuming a 4 msec print time and a 21 msec line feed time (which is within the state-of-the-art for line feed techniques and will also be equal to the cool down time for the print head) we have 4 msec + 21 msec = 25 msec per line. Therefore

$$\frac{1}{25\ \frac{\text{msec}}{\text{line}}} \times \frac{\text{msec}}{10^{-3}\text{sec}} \times \frac{60\text{sec}}{1\ \text{min}} = \frac{2400\ \text{lines}}{\text{minute}}$$

which is comparable with the speed of other type line printers of this size.

USAECOM has also sponsored a development of Miniaturized Techniques for Printing (MINIPRINT) with NCR. This is an exploratory development printer which prints at 15 characters/second, can be operated from a keyboard, and was developed for potential use as a self-contained, battery operated, tactical teleprinter. The package size of the present model (cigar-box size) may also contain all the required electronics. The printing is immediately visible on new thermal sensitive master paper. The master and first copy are made simultaneously in this printer and at least ten legible copies can be obtained using the master copy and plain paper in an off-line pressure device.

CONCLUSION

The Thermal Printer can print data received at any mixed rate of input — including input from a keyboard. The rate of input has been tried and tested at rates as high as 240 characters/sec which is certainly not the upper limit. Since we also get an electronic, non-impact printer which appears to be very reliable and can produce multiple copies on plain paper, thermal printing may have potential application in many systems — both communications systems and automatic data processing systems.

REFERENCES

1 W H PUTERBAUGH S P EMMONS
A new printing principle
Proceedings of the Spring Joint Computer Conference pp 121-124 1967

2 H SCHROEDER
Thermal printing
NCR Report AD-1232 pp 9-11 1964

Solid state synchro-to-digital converter

by G. P. HYATT
Teledyne Systems Company
Hawthorne, California

INTRODUCTION

The application of digital computers has been severely limited in systems that must interface with analog computer equipment. This limitation is particularly prevalent in aerospace systems where the lack of the appropriate interface equipment often precludes the use of a digital computer. Although digital computers are highly miniaturized and economical, the interface equipment often offsets these advantages with mechanizations that can exceed the computer size and cost. In order to enhance the applicability of digital computers for aerospace applications, Teledyne has expended considerable effort in the development of hybrid interface equipment. This hybrid equipment is mechanized with the latest solid state components and advanced packaging techniques to implement a universal interface that is miniaturized, modular, and economical. This interface unit is compatible in performance and packaging with the Teledyne family of digital computers.

The basic guidelines for the development of the hybrid interface equipment were to:

1. Place the burden of the conversion on the digital equipment,
2. Minimize the performance requirement placed on the analog equipment, and
3. Eliminate electromagnetic and electromechanical equipment.

The most complex part of the hybrid interface equipment is the synchro-to-digital (S/D) converter because of the three wire (120° coordinate system) AC analog form of the synchro signals. The Solid State S/D Converter accepts these three wire AC waveforms, performs a minimum of signal processing in the analog domain, then efficiently converts the data to a digital form for processing in the digital domain. The analog equipment converts the AC analog 120° coordinate signal forms to incremental digital orthogonal coordinate data. The digital data is processed with a digital "follow-up servo" implemented with Digital Differential Analyzer (DDA) computer modules. The DDA will insure scale factor and phase angle precision and automatically compensate for many errors introduced in the analog domain. The digital output of the S/D converter will contain the sine and cosine functions in addition to the angle. This is a significant advantage over converters that only generate the angle, because the trigonometric functions of the angle are often the parameters that are required.

Contemporary S/D converters perform the conversion with electromechanical follow-up servos driving digital shaft encoders or with switching of taps on transformer type electromagnetic equipment. These techniques require large, heavy, and expensive electromechanical or electromagnetic hardware used in a servo loop to null the input signal. The size, weight, performance, and cost considerations preclude the use of these converters for many aerospace applications.

A description of the Solid State S/D Converter mechanization will be presented following an analysis of the pertinent characteristics of the synchro waveforms.

Synchro signal forms

A synchro is an analog angular position transducer, where the output signal is an AC voltage with the amplitude indicative of the angular position of the rotor.

A schematic representation of a synchro is presented in Figure 1. The rotor is excited with an AC voltage that couples to the Y windings of the stator through transformer action. The angular position of the rotor, with respect to the stator, will determine the electromagnetic coupling between the rotor and each of the stator windings, thereby defining the amplitudes of the signals in the three windings. The stator output voltages are either in-phase or 180° out-of-

phase with the reference excitation. The amplitude and phase (in/out) of the output voltages is indicative of the angular displacement. The output signals are in time phase with the excitation signal, with the amplitude of the output signals conveying the trigonometric information. Phase shifts between the reference and the output waveforms do not convey angular information, but are potential sources of error in interpreting the output signals.

The excitation signal, shown in equation (1), is electromagnetically coupled to the output windings of the synchro, generating the output signals shown in equations (2) through (4).

$$V_{excitation} = E \sin \omega t \quad (1)$$
$$V_{0-1} = K_1 E \sin \omega t \sin\theta \quad (2)$$
$$V_{0-2} = K_1 E \sin \omega t \sin (\theta - 120°) \quad (3)$$
$$V_{0-3} = K_1 E \sin \omega t \sin (\theta - 240°) \quad (4)$$

The time dependent term, sin ωt, is common to all signals and is not a function of rotor position. The other terms of the synchro output equations are amplitude defining qualities, independent of the time varying portion of the waveform. It can be seen that the output signals are all in time phase with the excitation signal, and the amplitude is a trigonometric function of the angular position of the rotor. Equations (2) through (4) define the voltage induced in the winding from the common point of the three coil Y-connection to the output terminal. The common point of the three windings is usually not available for synchro follow-up operation. Therefore, it is necessary to operate on the voltages across the three output winding legs. The output signals of the synchro are listed in equations (5), (6) and (7). The subscripts define the terminals across which the voltages are measured.

$$V_{1-2} = K_2 E \sin \omega t \sin (\theta - 150°) \quad (5)$$
$$V_{2-3} = K_2 E \sin \omega t \sin (\theta + 90°) \quad (6)$$
$$V_{3-1} = K_2 E \sin \omega t \sin (\theta - 30°) \quad (7)$$

The signals defined in equations (5), (6) and (7) represent those signals which are available at the interface. Equations (5) and (7) contain sufficient information to completely define the synchro position, while equation (6) is redundant. The information contained in equations (5) and (7) can be organized in a more intuitive form with the use of the trigonometric identities, illustrated as equations (8) and (9).

$$\sin(\theta - 150°) = \sin\theta\cos 150° - \cos\theta\sin 150° \quad (8a)$$
$$= -\sqrt{3}/2 \sin\theta - 1/2 \cos\theta \quad (8b)$$
$$\sin(\theta - 30°) = \sin\theta\cos 30° - \cos\theta\sin 30° \quad (9a)$$
$$= \sqrt{3}/2 \sin\theta - 1/2 \cos\theta \quad (9b)$$

substituting equations (8b) and (9b) into equations (5) and (7) yields equations (10) and (11), respectively.

$$V_{2-1} = \frac{K_2 E}{2} \sin \omega t (\sqrt{3} \sin\theta + \cos\theta) \quad (10)$$

$$V_{3-1} = \frac{K_2 E}{2} \sin \omega\tau (\sqrt{3} \sin\theta - \cos\theta) \quad (11)$$

From equations (10) and (11), it can be seen that the voltages measured from synchro outputs 2 and 3; using synchro output 1 as a reference; is composed of sine and cosine components of the angular displacement of the synchro rotor. The coefficients of the corresponding trigonometric functions of θ from equations (10) and (11) are equal. Therefore, algebraic manipulation of equations (10) and (11) can be used to isolate the components of the interface signals that define the sine and the cosine of the synchro angular position. The addition and subtraction of equations (10) and (11) will yield equations (12) and (13), respectively.

$$V_{2-1} + V_{3-1} = K_2 \sqrt{3} E \sin \omega t \sin\theta \quad (12)$$
$$V_{2-1} - V_{3-1} = K_2 E \sin \omega t \cos\theta \quad (13)$$

The time varying term, sin wt, is removed with a Phase Sensitive Demodulator. This term is removed from the signals represented by equations (10) and (11) before they are added to form the signals represented by equations (12) and (13).

Equations (12) and (13) completely define the angular position of the synchro. It should be noted that the trigonometric functions of θ contribute only to the amplitude of the respective signals, introducing no inherent time or phase sensitive terms. The two signals defined in equations (12) and (13) are time coincident with the excitation to the synchro, excluding error mechanisms.

The resolver is an alternate type of angular position transducer sometimes used in place of the synchro. The resolver is shown schematically in Figure 1B. This device will transform the excitation signal into sine and cosine components of the angular displacement of the rotor. With the excitation, defined in equation (14), applied as illustrated in Figure 1; the output signals from the resolver will be as defined in equations (15) and (16).

$$V_{excitation} = E \sin \omega t \quad (14)$$
$$V_s = E \sin \omega t \sin\theta \quad (15)$$
$$V_c = E \sin \omega t \cos\theta \quad (16)$$

These equations are similar in form to equations (12) and (13), yielding a common signal form upon which to develop compatible synchro/resolver-to-digital converters.

A Scott-T transformer is a device that is capable of converting between two phase and three phase signal forms. The synchro signals are in three phase form while the resolver signals are in two phase form. Therefore, the Scott-T is a convenient conversion

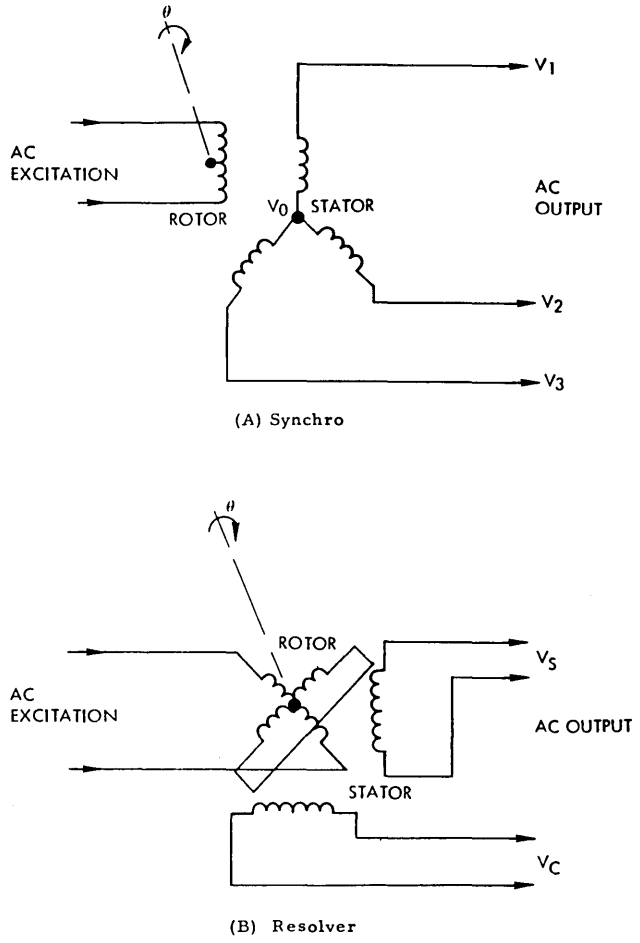


Figure 1 – Angular position transducer

device between synchro and resolver type signals. It is desirable to have the signals in two phase form for convenience of processing and conversion. Therefore, the Scott-T may be used to convert the synchro signals to two phase form. Most contemporary synchro-to-digital converters use a Scott-T to convert the synchro signals to two phase form prior to performing the digital conversion. A Scott-T is a relatively bulky, heavy, and expensive component. Therefore, the techniques described for the Solid State S/D Converter perform the conversion between the two and three phase signal forms using semiconductor components instead of the transformer type Scott-T. This conversion is implemented merely by adding summing resistors to the input of an integrated circuit operational amplifier, thereby implementing equations (12) and (13). It is obviously a good tradeoff to replace a Scott-T with only two summing resistors.

Synchro-to-digital converter

A representative mechanization of the Solid State S/D Converter is illustrated in Figure 2. It is basically a succession of conversions from each signal form to a more convenient signal form that permits appropriate processing. This approach permits many variations of the conversion concept, depending upon interface considerations and tradeoffs. In a typical application, the synchro transmitter is excited from a reference AC voltage supply. The excitation is coupled to the output windings as a function of the mechanical angular displacement of the synchro rotor. This angular displacement is illustrated as the θ_I input angular displacement. Three output lines are presented to the interface with signals that are indicative of the angular position of the synchro. These interface signals are in AC analog form, with an AC carrier frequency that is amplitude modulated as a trigonometric function of the angular displacement of the synchro. The time varying components of these signals are either in-phase or 180° out-of-phase with the reference excitation. A Phase Sensitive Demodulator (PSD) is used as an AC analog to DC analog converter. The Demodulator is a synchronous chopper, switching the input signals onto the output line in synchronism with the reference signal waveform. The output of the Phase Sensitive Demodulator is a DC signal with a high ripple content, similar to a full wave or half wave rectified waveform, whichever is applicable.

In typical applications of a Phase Sensitive Demodulator, the output signals are filtered to remove the large ripple content. In the mechanization illustrated in Figure 2, the output of the Phase Sensitive Demodulator is operated on by a Reset Integrator (RI), which will provide a filtering function superior to that of a passive filter. In addition, the ripple will not propagate into system errors due to the inherent error compensation characteristics of this converter.

The Reset Integrator provides the primary function of a DC analog to pulse rate converter. In the mechanization shown, the Reset Integrator also provides the secondary function of algebraic summation of the synchro information, described analytically in equations (12) and (13). This summation function effectively converts from three phase synchro type information to two phase resolver type information. Therefore, the output of the Reset Integrator is in orthogonal coordinate trigonometric functions.

The Reset Integrator primary function of DC analog to pulse rate conversion is accomplished by implementing an analog integrator which integrates the DC voltages applied at the input. When the integrator output exceeds a voltage threshold, a precise reset pulse is generated to reset the integrator a calibrated amount by discharging the feedback capacitor. The rate at which the integrator continues to exceed the

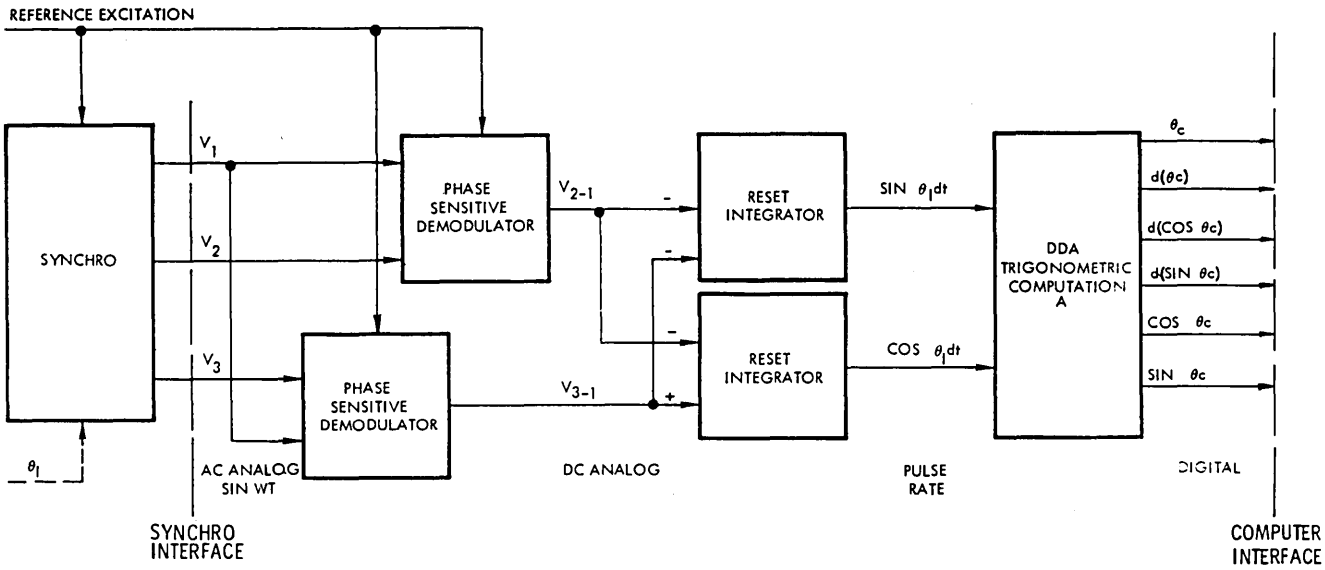


Figure 2—Synchro-to-digital converter

threshold is a function of the input voltage levels. The resetting pulse rate is also used as the Reset Integrator output and is indicative of the rate at which the integrator continues to exceed the output voltage threshold. Therefore, the output pulse rate is directly proportional to the average input voltage magnitude.

The mechanization illustrated in Figure 2 makes multiple use of the Reset Integrator functional block, where it is used to:

1. Perform the DC voltage to pulse rate conversion.
2. Perform the three phase synchro to two phase resolver type signal conversion.
3. Perform the filtering function for the DC signals from the Phase Sensitive Demodulator.

The ease with which the multiple functions are implemented in the analog domain significantly reduces handware and improves accuracy of this mechanization. For example, the three phase synchro to two phase resolver type signal conversion is accomplished by the addition of one resistor to each Reset Integrator summing junction. These two resistors are used to replace the Scott-T transformer type three phase to two phase signal converter. Therefore, the advantages of this technique is quite significant.

The Reset Integrator outputs are pulse trains, the rate of which is proportional to the trigonometric functions of the synchro angular displacement. These two pulse trains are significantly amplitude sensitive; where synchro excitation variations, synchro transformation ratios, and other mechanisms that affect the scale factor of the signals will affect the amplitude of the Reset Integrator output. Direct use of this pulse rate information in the computation will result in amplitude or scale factor errors, but which will appear as angular displacement errors. These scale factor errors can be conveniently eliminated by using the ratio of the two pulse trains instead of each pulse train by itself to determine the angular displacement of the synchro.

A Digital Differential Analyzer (DDA) trigonometric computational block is implemented to eliminate the scale factor sensitivity of the synchro signals. This computation is described in the corresponding section of this report. Effectively, equation (17) is implemented with a DDA computational block.

$$\sin(\theta_I - \theta_C) = \sin\theta_I\cos\theta_C - \cos\theta_I\sin\theta_C \qquad (17)$$

A DDA sin-cos generator, used as a "digital resolver," is rotated computationally to balance the input pulse rates. The solution of equation (17) using DDA techniques assures the availability of the synchro trigonometric functions that are completely independent of any scale factor type errors from the input. The outputs of the computation block are whole number serial trigonometric (sine and cosine) functions, whole number serial angular information, incremental trigonometric functions, and incremental angular information. This DDA computation is a necessary part of this Synchro-to-Digital Converter concept, because the scale factor sensitivity of the

synchro information would not permit sufficient accuracy, in most applications, without this compensation. As a contrast, the insensitivity of the converter outputs to the input scale factor parameter significantly decreases the cost of the analog equipment and provides a very significant amount of automatic compensation of scale factor type errors in the analog hardware. The predominating error mechanisms in the analog equipment are of a scale factor nature. Therefore, the elimination of the scale factor sensitivity consideration will permit extremely accurate analog mechanizations with only moderate consideration to many of the predominating error mechanisms. In addition, this DDA computation generates the trigonometric functions of the angle in addition to the angular information. Other types of S/D Converters typically generate the angular information only, requiring additional computation to generate the trigonometric functions of the angle. Therefore, cost and equipment comparison between this mechanization and other converters should be made on a comparable basis, where the DDA computation is included in the comparison only if the alternate converter will present the trigonometric functions of θ at the computer interface. If the other converter will generate only the angular information, a functional block equivalent to the DDA computation must be added to generate the trigonometric functions of the synchro angle. An alternate tradeoff would be to draw the computer interface at the input to the DDA computational block for a realistic comparison with the alternate type of converter. That is not to indicate that this converter can operate independent of the DDA computational block, but only to compare the alternates on an equivalent basis. These considerations are contingent upon the requirement for trigonometric functions of the angle in the computation. Experience has shown that, in general, the trigonometric functions of the angle are required and not the angle explicitly.

An alternate mechanization of the Synchro-to-Digital Converter is illustrated in Figure 3. This mechanization accepts the three phase synchro type information and converts it to two phase resolver type information with a Scott-T transformer. The two channels operate virtually independently, without the need for cross summing DC analog signals. The outputs of the Scott-T transformer are AC waveforms with amplitudes indicative of the trigonometric functions of the synchro angular position. The Phase Sensitive Demodulators will act as AC analog to DC analog converters, with the Reset Integrators acting as DC analog to pulse rate converters. The DDA computation is identical to that used in the previously described mechanization. The considerations for this alternate mechanization are identical to that for the first mechanization, with the exception of the added cost and weight factors introduced by the Scott-T transformer.

DDA trigonometric computation

The DDA trigonometric computation functional block is a requirement for this Synchro-to-Digital Converter mechanization in order to eliminate the scale factor sensitivity of the converter. It is conceivable that, in certain types of applications, the synchro scale factor parameters could be controlled

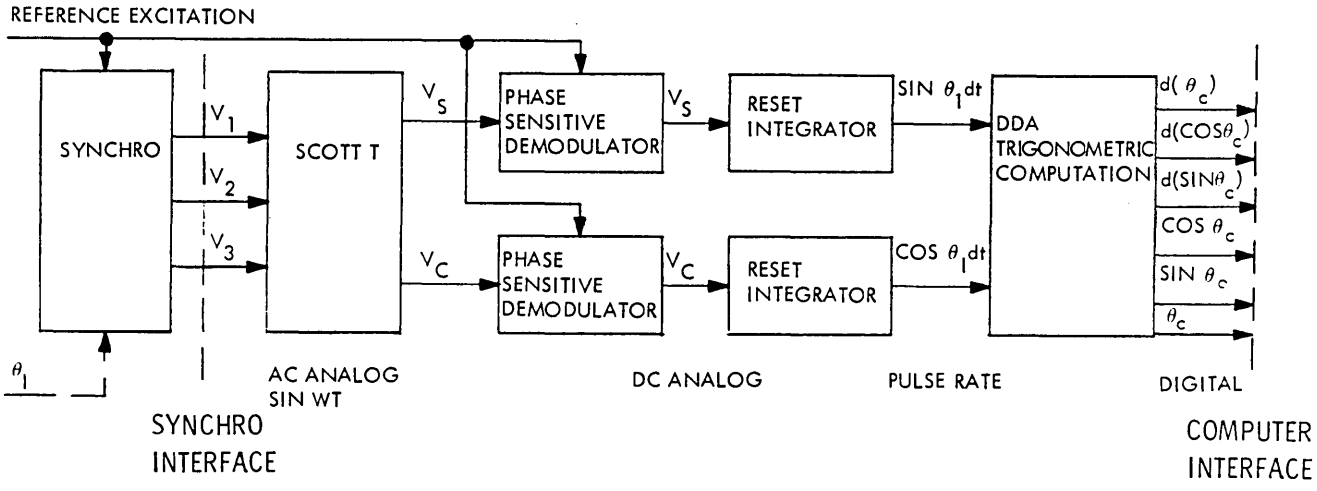


Figure 3 – Synchro-to-digital converter

within the requirements of the conversion. In the general case, tolerances are an order of magnitude greater than the required conversion accuracy, necessitating this computation. For high accuracy type converters, the DDA equipment that is required will be offset by the simplification of the analog equipment due to the reduction of the scale factor sensitivity of the converter.

The converter implicit servo is implemented in the digital domain with Digital Differential Analyzer (DDA) computational elements. This DDA is a parallel computation serial word computer; where all computations are performed simultaneously in a bit by bit serial fashion. The DDA performs computations by successive additions accomplished at the rate of 32,000 iterations/second to approximate integration or multiplication. A detailed description of DDA operation is contained in Reference 1. The DDA Computation is illustrated functionally in Figure 4. This functional diagram implements equation (17), which is the trigonometric identity for the sine of the difference of two angles. The input angle, θ_I, is defined by the synchro rotor displacement with respect to the stator. The computed angle, θ_c, is contained in the DDA sin-cos generator. For the condition that θ_I is equal to θ_c, equation (17) will be nulled. The implementation of equation (17) with a DDA in an implicit servo type function will cause the DDA sin-cos generator to be driven to a condition equivalent to the angular displacement of the synchro. As the synchro is rotated, the DDA sin-cos generator will be servoed to the corresponding angular position. Effectively, the DDA Computation block performs the function of a digital "follow-up servo."

The functional diagram, illustrated in Figure 4A, implements an implicit servo to solve equation (17). DDA computational elements 1 and 2 are used as pulse rate multipliers, where the input trigonometric function pulse rate information is multiplied by orthogonal trigonometric functions from the "digital resolver." The symbolism used for DDA computational elements 1 and 2 indicates that the R register and R logic functions are utilized, but the Y register word is fanned-in from other DDA computational elements. For example, the Y register number for computational element 1 is obtained from the Y register of computational element 4. This Y register number is fanned-out to the two R logic functions, one in computational element 4 and the other in computational element 1. This fan-out is illustrated more graphically in Figure 4B, the sub-functional block diagram. Computational elements 3 and 4 implement a DDA sin-cos generator. The two pulse rate products from computational elements 1 and 2 are subtracted in the rate summer to generate an error rate which is a solution to equation (17). If this error rate is zero, indicative of the two trigonometric products being equal, the "digital resolver" is at the equivalent angular position of the synchro. If the error rate is not at null, the incremental pulses will be used as the $d\theta$ inputs to the DDA sin-cos generator, resulting in this "digital resolver" being rotated to null the error rate.

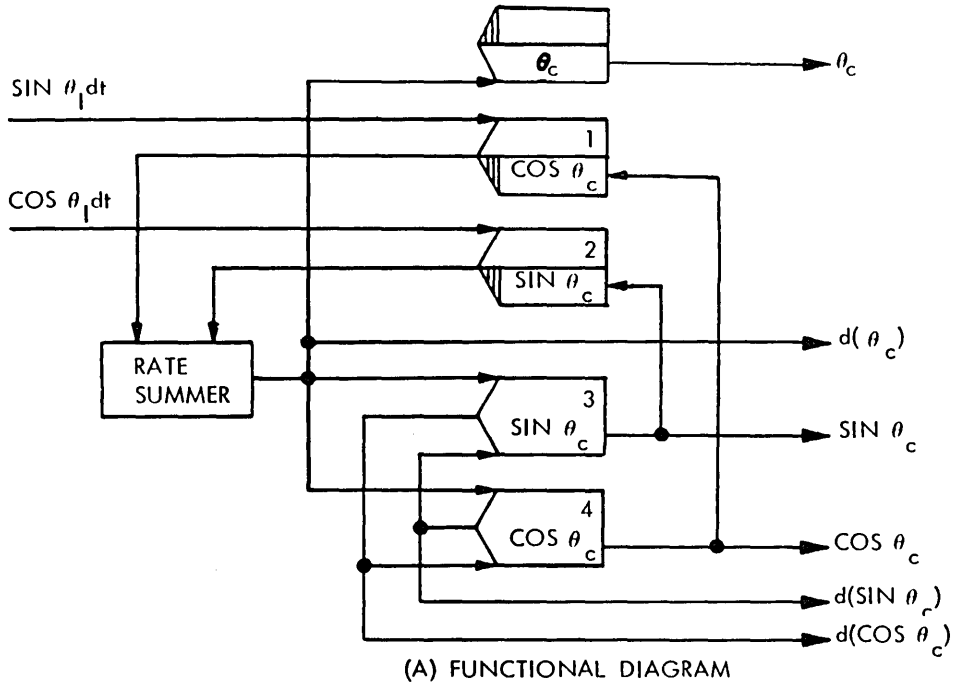
The sub-functional block diagram, illustrated in Figure 4B, implements rectangular integration for the pulse rate multipliers and trapezoidal integration for the sin-cos generators. The trapezoidal integration algorithm will reduce the error buildup in the "digital resolver" to an extremely small level.

It can be seen that scale factor coefficients of the trigonometric functions will not affect the computation at the "digital servo" summing junction, which is the rate summer. A true scale factor coefficient will be common to each of the trigonometric sub-products of equation (17), thereby affecting the gain of the "digital servo" but not the null. The DDA computation that implements equation (17) will be nulled independent of the scale factor of the respective angular functions.

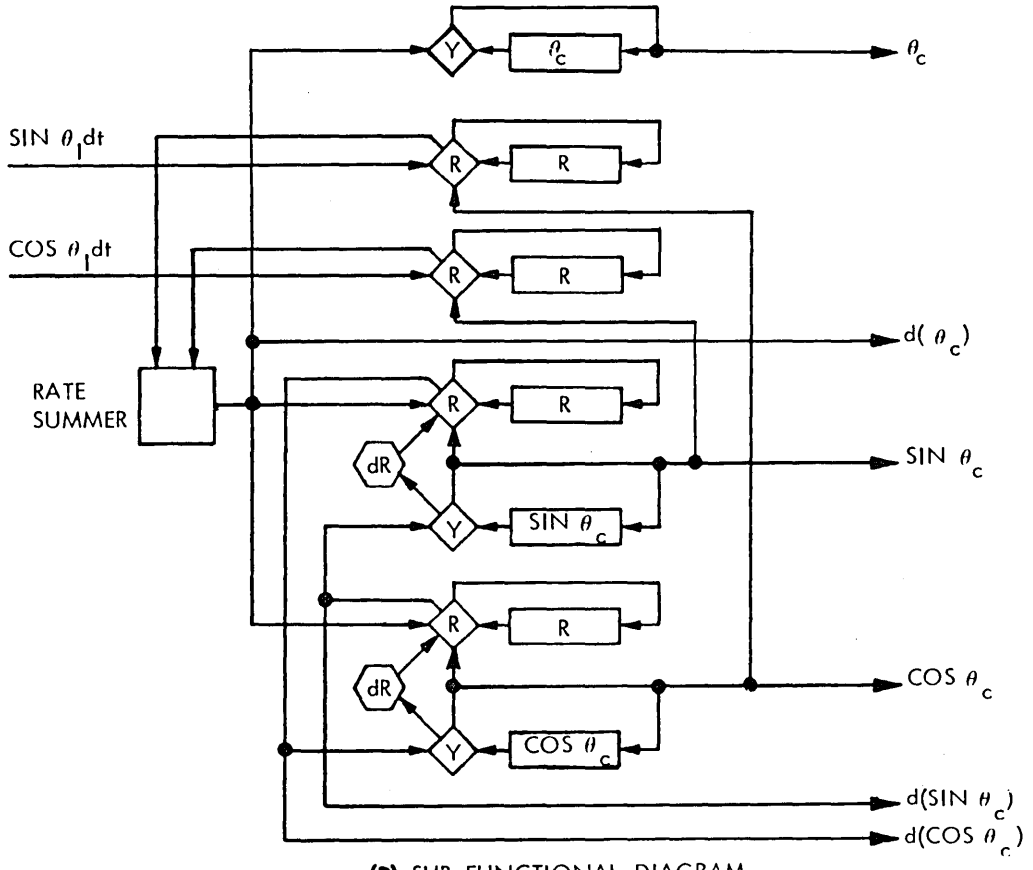
The output scale factor of the "digital servo" is dependent on the initial conditions loaded into the sin-cos generator. The vector sum of initial conditions for the sin-cos generator will define the scale factor of the output trigonometric functions. These initial conditions will be simple to generate, permitting a zero to be loaded into the sin θ_c register and a nominal scale factor, typically unity, to be loaded into the cos θ_c register. These initial conditions are representative of an initial 0° angular position with a unity scale factor. After the initial conditions are loaded, the "digital servo" loop will be closed; thereby permitting the "digital resolver" to be driven to the corresponding angular position of the synchro. Therefore, only constant initial conditions need be loaded; as the "digital resolver" will automatically generate the proper parameters after the "digital servo" loop has been closed.

Phase sensitive demodulator (PSD)

The Phase Sensitive Demodulator is effectively a synchronous chopper that switches the input signals in phase with the reference signal. Because the input signals are either in phase or 180° out of phase with the reference signal, a synchronous chopper will have the effect of rectifying these signals, thereby generating an equivalent DC voltage proportional to the amplitude of the AC voltage. The output waveform will be rectified positive or negative voltage levels. The PSD is implemented with FET (Field Effect



(A) FUNCTIONAL DIAGRAM



(B) SUB-FUNCTIONAL DIAGRAM

Figure 4—DDA trigonometric computation

Transistor) switches, synchronously chopping the AC waveform into the input of an operational amplifier, illustrated in Figure 5.

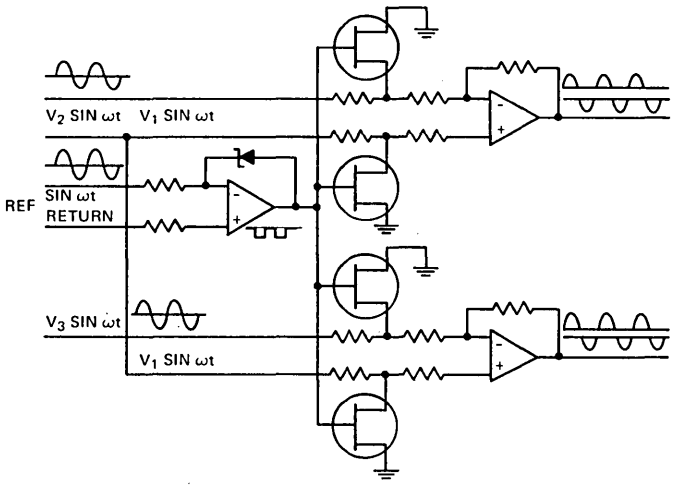


Figure 5 – Phase sensitive demodulator

The PSD exhibits high common mode rejection characteristics, which are inherent in the operational amplifier inputs. Each channel of the PSD is driven by the reference amplifier, which is used to "square-up" the reference input. This signal is used to switch or synchronously chop the input signals with FET shunt switches. The two switches for each channel are driven in phase, controlled by the output of the reference amplifier. A half wave PSD results, where the output has an amplitude and polarity determined by the algebraic difference of the two differential input signals. A half-wave Demodulator is acceptable due to:

a. The freedom to achieve large scale factors through the inherent gain of the operational amplifiers.
b. The lack of concern with filtering the high ripple content.

Many of the analog errors are automatically compensated in the digital equipment. The FET switch on-resistance introduces an insignificant error due to the swamping effect of the input resistors. Considerations such as signal phase shift and ripple do not propagate to system errors, due to the automatic compensation characteristics of the DDA.

Reset integrator (RI)

A Reset Integrator is a functional block that generates a pulse rate output proportional to a DC voltage input. It is used as a DC analog to pulse rate converter. This function is implemented by using an analog integrator to generate a ramp output, the slope of which is proportional to the DC voltage input. A threshold detector generates a reset pulse when the ramp exceeds a fixed threshold. The reset pulse will reset the ramp by a fixed increment and generate an output increment. The threshold detector, by generating reset pulses, will maintain the ramp amplitude below the fixed threshold. The rate at which the ramp continues to exceed the threshold, in the presence of the reset pulses, is indicative of the average DC voltage impressed at the input. Therefore, the pulse rate is proportional to the input DC voltage.

A typical Reset Integrator is illustrated schematically in Figure 6. This Reset Integrator is of the ternary type, generating output pulses on the plus or minus

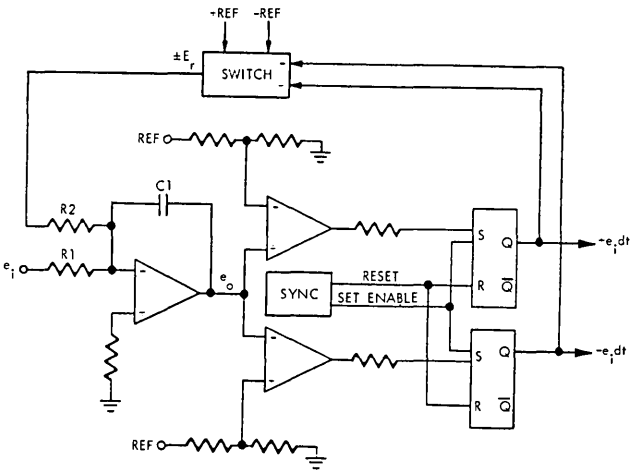


Figure 6 – Reset integrator

lines depending upon the polarity of the input DC voltage. The input voltage, e_i, is integrated with the operational amplifier integrator, where the output voltage, e_o, is equal to the integral of the input voltage. The integration time constant is R_1C_1, as illustrated in equation (18).

$$e_o = \int \frac{e_i dt}{R_1 C_1} \tag{18}$$

The output voltage is applied to two operational amplifier level detectors, analogous to the conventional schmidt triggers. These operational amplifiers will generate a large positive output when the output voltage exceeds the voltage threshold defined by the resistor voltage divider. The positive and negative inputs to the summing junction are used to achieve bipolar threshold detection. When the output of the Reset Integrator exceeds a threshold, the output of one threshold detector will go positive, thereby enabling the flip-flop to be set. The synchronization source will cause that flip-flop to be set at the appro-

priate computer bit time, yielding a pulse-on-demand type of Reset Integrator. This synch source will also provide the reset pulse for the flip-flop, providing precise timing for the pulse width. The output of the flip-flop will be high for one pulse width period, generating an output to the computer and a feedback signal to the switch. The switch will cause a reference voltage to be applied to the integrator summing junction through the reset gain setting resistor, R_2. The polarity of the reference voltage applied to the operational amplifier input is a function of the threshold detector whose threshold was exceeded. The reset voltage will be integrated, thereby driving the output voltage below the threshold. The reset pulse is a precise increment, defined in equation (19).

$$\Delta e_o = \int_o^{t_1} \frac{E_r dt}{R_2 C_1} \tag{19a}$$

$$\Delta e_o = \frac{E_r t_1}{R_2 C_1} \tag{19b}$$

In normal operation, the integrator output, as reset by the feedback pulses, will have a value defined in equation (20).

$$e_o = \int_o^t \frac{e_i dt}{R_1 C_1} - N\Delta e_o \tag{20}$$

The parameter N is the number of pulses out of the Reset Integrator. The output pulse train from the Reset Integrator is the most significant part of the solution. The integral term, which is the least significant part of the solution, will be retained in the output of the integrator. Deleting the integral term from equation (20) and substituting equation (18) and (19) into the balance of equation (20) will result in the transfer function for the Reset Integrator. The integral term can be deleted because it is the remainder which is less than the resolution of the Reset Integrator. Differentiating equation (20) with respect to time, then rearranging will yield equation (21); which defines the output pulse rate in terms of the input voltage. Equation (21) is the transfer function for the Reset Integrator.

$$\frac{dn}{dt} = \frac{e_i R_2}{R_1 E_r t_1} \tag{21}$$

All of the terms in this equation define the scale factor of the functional block. Error components inherent in these terms will cause a scale factor type error. Because this converter mechanization is insensitive to scale factor type errors common to both channels, most of these error mechanisms will not contribute to the overall converter error. This is a very significant error compensation affect, because much of the complexity in mechanizing the Reset Integrator is expended to minimize scale-factor errors. This complexity includes precision reference voltage supplies, low temperature coefficient resistors, precision switching envelopes, and other such considerations.

Hardware description

The electronics industry has made tremendous progress in the development of advanced electronic component and techniques. In particular, the advent of integrated circuits has virtually revolutionized the electronics industry, especially the digital computer area. The trend has been towards components that are very small, fast, low in power consumption, and highly reliable. In order to take advantage of the new hardware available, new and sophisticated handling and packaging techniques have been required. In order to handle and package the miniature components, much of the advantage of the small size has been lost. Teledyne has developed packaging techniques that make maximum use of the characteristics of the advanced components. In particular, the Micro-Electronic Modular Assembly (MEMA) takes maximum advantage of the characteristics of integrated circuits. These are:

a. Preserving the small-sized characteristic of the integrated circuit chip by placing many chips in a single package.

b. Preserving the high speed characteristics of the integrated circuits by placing them in very close proximity with extremely short interconnections.

c. Preserving the inherent reliability of the integrated circuit by eliminating multiple packaging levels.

In addition, greatly improved manufacturing and maintenance techniques are realized and manufacturing costs are significantly reduced. The basic MEMA is a hermatically sealed flat pack with 24 leads. The dimensions are $1.0 \times 0.75 \times 0.06$ inches. Each MEMA can contain up to 30 digital integrated circuit "bare chips" or 25 analog chips (integrated circuit, resistor, capacitor, etc.). The digital MEMA is illustrated in Figure 7 and the analog MEMA is illustrated in Figure 8. The "bare chips" and 1 mil wire jumpers are clearly visible with the covers removed. The interconnection pattern is photo-etched onto the ceramic substrate. The chips are die-bonded to the substrate, then the jumpers are used to connect the chip signal pads to the substrate interconnection pattern. The jumpers are composed of 1 mil (1/1000 inch) diameter aluminum wire ultrasonically bonded to the chip substrate.

The MEMA contains the equivalent functional complexity of a large printed circuit board within a highly miniaturized package; resulting in enhanced performance, reliability, cost, size, and weight.

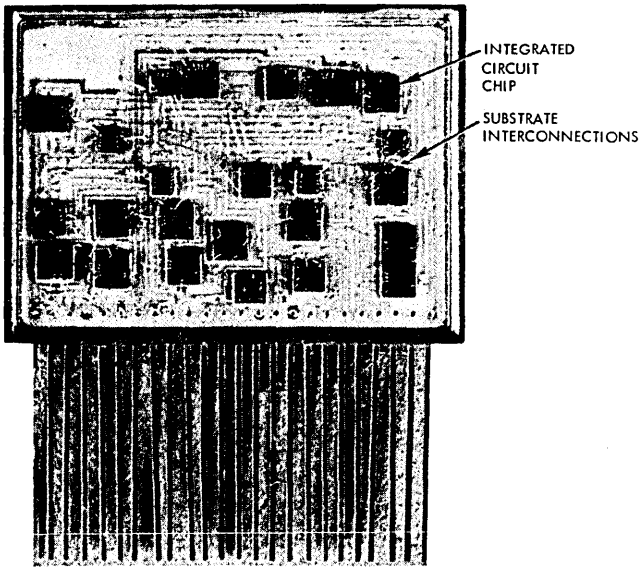


Figure 7 – Digital circuit MEMA

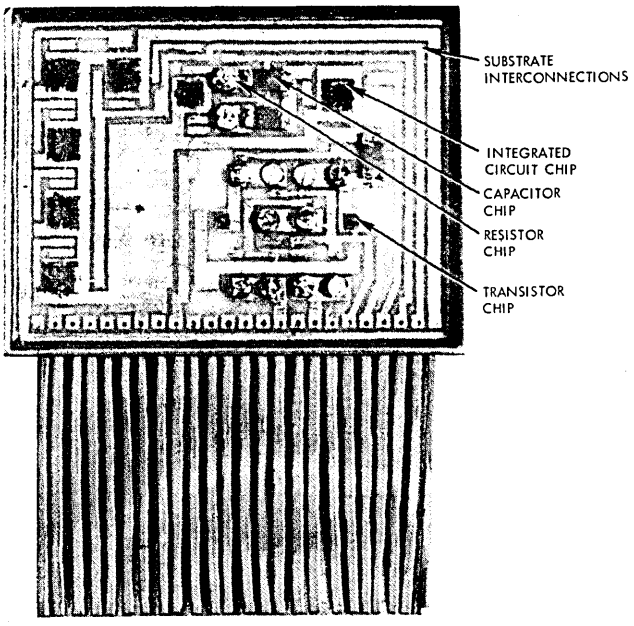


Figure 8 – Analog circuit MEMA

Error considerations

As previously described, scale-factor type errors common to both channels of this converter will not propagate to system errors, but will be compensated for automatically. Most of the temperature sensitive errors, voltage sensitive errors, phase shift errors, reset pulse shape errors, and many other types, will be automatically eliminated due to the cancellation nature of the amplitude sensitive errors. Many of the balance type errors; such as resistor and capacitor tolerances, bias errors, and other types; can be minimized by simple adjustments on one channel with respect to the other. These errors will then have a tendency to track each other over temperature and voltage type variations, providing automatic compensation over the operating spectrum. These automatic compensations and simple balance type adjustments very significantly decrease the cost and complexity of the converter while significantly increasing the inherent and ultimate accuracies of the device.

The Reset Integrator is the more critical of the two major analog functional blocks in the converter. Therefore, specific errors will be discussed for the Reset Integrator with reference to Figure 6. The Reset Integrator, when used alone, is extremely sensitive to variations in the reference voltage that defines the reset pulse. This reference voltage will be common to both Reset Integrators and, therefore, will contribute corresponding scale-factor errors to both channels. Therefore, a precision reference supply is not required for this converter mechanization. The voltage may be permitted to vary within moderately large tolerances over the operational and environmental extremes. The ripple content of the PSD output will have a tendency to frequency modulate the Reset Integrator outputs. The ripple content for both channels is phase locked and of a scale factor nature. Therefore, this signal characteristic will not produce converter errors.

Some of the discrete resistors and the C_1 capacitor, illustrator in Figure 6, will not contribute errors to the Reset Integrator operation. The balance of the discrete resistors will contribute primarily scale-factor type errors. The tolerance of these resistors will cause a scale-factor spread between the two channels of the converter. This nominal spread will cause an error that will not be cancelled, but can be adjusted for. Operational and environmental variations that affect the value of the resistors will be common to both channels of the converter and will be automatically compensated. Therefore, low thermal coefficient resistors are not necessary for this mechanization. The saturation resistance of the field effect transistor (FET) switches has a spread of values that can be swamped-out if the value of R_2 is made sufficiently large. Additional accuracy can be achieved by matching the FET switches in both channels. The temperature sensitivity of the FET parameters will be common to both channels and should introduce scale-factor type errors that will automatically compensate.

The Reset Integrator scale factor is inherently dependent upon the shape of the reset pulse. Because predictability of the Reset Integrator scale factor is not a prime consideration, the shape of this reset pulse need not be accurately defined geometrically. The only requirement is that both channels exhibit corresponding pulse areas. Therefore, switching devices need only have moderately fast switching characteristics.

An analysis of the AC signal inputs to the Phase Sensitive Demodulators verifies that a phase shift between the reference excitation and the synchro signals contribute a scale factor type of error to the Converter. Therefore, there will be no error contribution due to the gross phase shift thru the synchro. The system error will only be a function of the difference in phase shift between the two channels.

The types of errors described that are not of the automatic compensating nature are primarily of the bias type. Most of the bias errors can be balanced with a single padding resistor in one of the Reset Integrator channels. This simple balance technique would reduce the nominal bias and matching errors, leaving the errors caused by changes in operational and environmental conditions as the predominating effects. Most of the operational and environmental sensitive terms will be common to both channels and of a scale factor type nature. Therefore, only the second order differences in operational and environmental sensitivities between corresponding components and sensitivities of bias type effects will contribute to the converter errors. Because of these considerations, it is relatively easy to reduce the predominating error mechanisms inherent in the converter to those normally considered as third order type affects.

This description of the predominating errors in the Converter illustrates that there is little need for a highly precise reference power supply, low temperature sensitive equipment, or sophisticated circuit design techniques to facilitate a highly accurate Reset Integrator. Relatively simple design techniques coupled with a simple balancing scheme could easily implement highly accurate analog equipment for this Converter.

CONCLUSION

The Solid State S/D Converter permits the mechanization of a low cost accurate, versatile, and miniature interface that can accommodate synchro input signals. This Converter will increase the feasibility of using digital computers in conjunction with electromechanical analog computers, which are common in aerospace applications. In addition, a more optimum mix of analog and digital techniques will be practical for hybrid computers.

REFERENCES

1 E L BRAUN
Digital computer design
Academic Press New York Chap 8 p 448 1962

2 S A DAVIS B K LEDGERWOOD
Electromechanical components for servomechanisms
McGraw Hill Book Company Inc New York Chap 3 p 92 1961

3 B C KUO
Automatic control systems
Prentice-Hall Inc New Jersey chap 4 p 92 1962

A new high-speed general purpose I / O with real-time computing capability

by KENNETH FERTIG
DUNCAN B. COX, JR.
MIT Instrumentation Laboratory
Cambridge, Massachusetts

INTRODUCTION

Real-time data acquisition and control systems incorporating a general purpose digital computer (GPC) are considered for convenience to be composed of three parts: transducers and transmission paths, an input-output mechanism (I/O), and the GPC. The transducers and transmission paths considered in particular are those resulting in the desired data being phase-modulated on carrier waveforms. This type of phase-encoded information may be received from a variety of sources, among which are shaft-angle resolvers with sine-cosine excitations, and Doppler navigation systems. The primary focus of attention in this paper is a new I/O which can receive the phase-modulated waveforms directly, perform a variety of processing functions on the raw data in its phase-modulated form, and present the processed data in a convenient binary format to the GPC. The new I/O can process data from a number of sources in parallel at relatively high speeds, thereby leaving the GPC time for monitoring, adaptive parameter adjustment, and other sophisticated decision and control functions. Similar techniques utilizing different phases of the computer clock signal permit the GPC to generate digital and/or analog commands to transducers via the I/O.

Some examples of transducers and transmission links are briefly presented as background material and motivation for a discussion of the I/O itself. The operation of the I/O is then discussed in detail with reference to a practical system for obtaining and processing whole-angle data from a multi-speed shaft-angle resolver and, in more generality, with reference to a wide variety of signal sources producing phase-modulated information.

The ease of performing analog/digital conversion (and its inverse) with phase information is the key element in the I/O discussion. As a result, the real-time control problem and hybrid computer problem may be considered as essentially the same time shared computer problem, and the traditionally complex I/O problem of tying analog elements to the GPC may be solved in the same simple way.

Several significant improvements can be made in computer usage by the proper design of an I/O. In particular, the programming can be simplified, more useful computing can be carried out in a given time interval, and greater system flexibility can be achieved.

Transducers and transmissions links

In preparation for the detailed discussion of the I/O, some examples of transducers and transmission links producing phase-encoded information are given. These examples serve to introduce some simple but useful mathematical notation and to provide a physical interpretation of the origin and meaning of the phase-encoded waveforms that are central to the discussion of the new I/O. The examples also serve to introduce the ideas that there may be several information sources (either independent or dependent) operating simultaneously and that the information can conveniently be multiplexed without altering the phase encoding. The new I/O has the natural capability of processing the phase-encoded information from several sources simultaneously.

A typical example of a transducer producing phase-modulated information is the shaft-angle resolver shown in Figure 1. These transducers are found in equipment ranging from inertial to machine-shop. The input signals e_1 and e_2 are any periodic waveforms with fundamental components $E \sin 2\pi f t$ and $E \cos 2\pi f t$,

respectively. The resolver produces an output

$$e_o = e_1 \cos \phi + e_2 \sin \phi \quad (1)$$

where ϕ is the mechanical resolver angle. Hence the fundamental component of the output is

$$\text{fund}\,(e_o) = E \sin (2\pi f t + \phi) \quad (2)$$

which is linearly phase modulated by the resolver angle. Henceforth, we shall assume for convenience that the input waveforms are square waves with frequency f and phase angles 0° and 90°, these are denoted by $f\underline{/0}$ and $f\underline{/90}$, respectively, as shown in Figure 1. Implicit in this notation is the assumption of a reference zero phase angle, which we shall assume is established by a reference clock.* The output of the resolver we shall denote by $f\underline{/\phi}$ to indicate that the fundamental component of the waveform is of frequency f and phase angle ϕ.

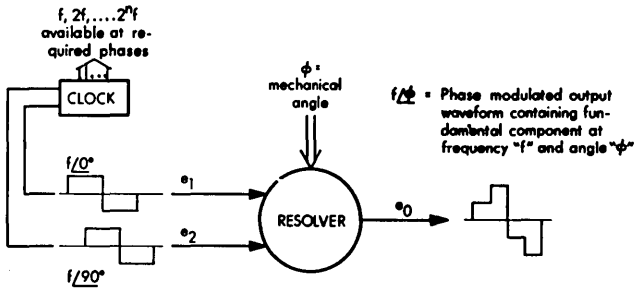


Figure 1—A shaft-angle resolver with phase-modulated output

Often an m-speed resolver** yielding an output $f\underline{/m\phi}$ may be used with a one-speed resolver on a common shaft to assign roughly equal parts (in the sense of double precision) of the desired angle information to each of two modulated waveforms. Where there are several resolvers, their output may be time multiplexed over a common transmission path with, e.g., four samples of each waveform taken per cycle, without distorting the phase of the fundamental component.[1] Another possibility is to effect frequency multiplexing by choosing the excitation frequencies for different resolvers as binary multiples, e.g., f, 2f, 4f, etc. Of course, both time and frequency multiplexing may be used simultaneously. In that case the outputs of the time multiplexer would be several xero-order-held waveforms, each resembling the resolver output waveform shown in Figure 1, but at different excitation frequencies. These phase-modulated outputs would be received by the new I/O. The frequency demultiplexing of the information is automatically achieved in the new I/O because an integral part of the new I/O is a set is a set of phase-locked loops.

Other examples of transducers yielding phase-modulated information abound: Loran, Doppler radar, sonar; etc. In these systems phase shift data correspond to measurements of distance. Two or more carrier frequencies (often multiplexed on a third) are often used to measure the same distance variable. This technique is completely analogous to that of using one-speed and m-speed resolvers to measure a common shaft angle.

The new I/O

A basic element in the new I/O is the phase-locked loop† shown in Figure 2a. It consists of a phase-sensitive detector (PSD), a low-pass filter (LPF) with transfer function F(s), a voltage-controlled oscillator (VCO),‡ and a binary n-stage forward counter (countdown). The countdown output $f\underline{/\theta}$ is a square wave which in normal operation is locked in frequency and phase with the fundamental component of the input $f\underline{/\phi}$ to the phase-locked loop. Ideally, θ is equal to ϕ plus 90°; careful loop design can insure that this relation is maintained reasonably well, in many cases to within a small fraction of a degree. The PSD can be a simple switching modulator that multiplies the input waveform by $+1$ or -1, depending upon the state of the countdown output. The LPF extracts the average value of the PSD output and, in its most general form, performs several other filtering functions. The basic phase-locking operation of the loop depends upon the action of the PSD in producing an error signal to increase or decrease the frequency of VCO oscillation in order to drive the phase error to zero. This mechanism is illustrated by the block diagram in Figure 2b. Because of the presence of the n-stage countdown, the VCO frequency is 2^n times the input frequency. Loops of this type have been operated satisfactorily with VCO frequencies as high as 5 mc and with as many as ten countdown stages. Standard heterodyning techniques can be used to accommodate input frequencies that are impractically high for the basic loop shown in Figure 2a.

The effect of the phase-locked loop is to create a set of square waves, each wave being the output of one stage of the countdown in Figure 2a, that track the phase ϕ of the fundamental component of the input waveform. The phase angle θ of this set with

*A clock is a high-frequency oscillator followed by a (typically binary) countdown chain from which several signals with locked frequencies and phases may be obtained .

**An m-speed resolver has windings with m pole pairs so that its output phase angle passes through 360 degrees when the mechanical shaft rotates through 360/m mechanical degrees.

†For good sources of information on the behavior, design and use of phase-locked loops the reader is referred to Tausworthe[2] and Gardner.[3]

‡Any simply controllable oscillator is usable, including digital oscillators made by DDA techniques.

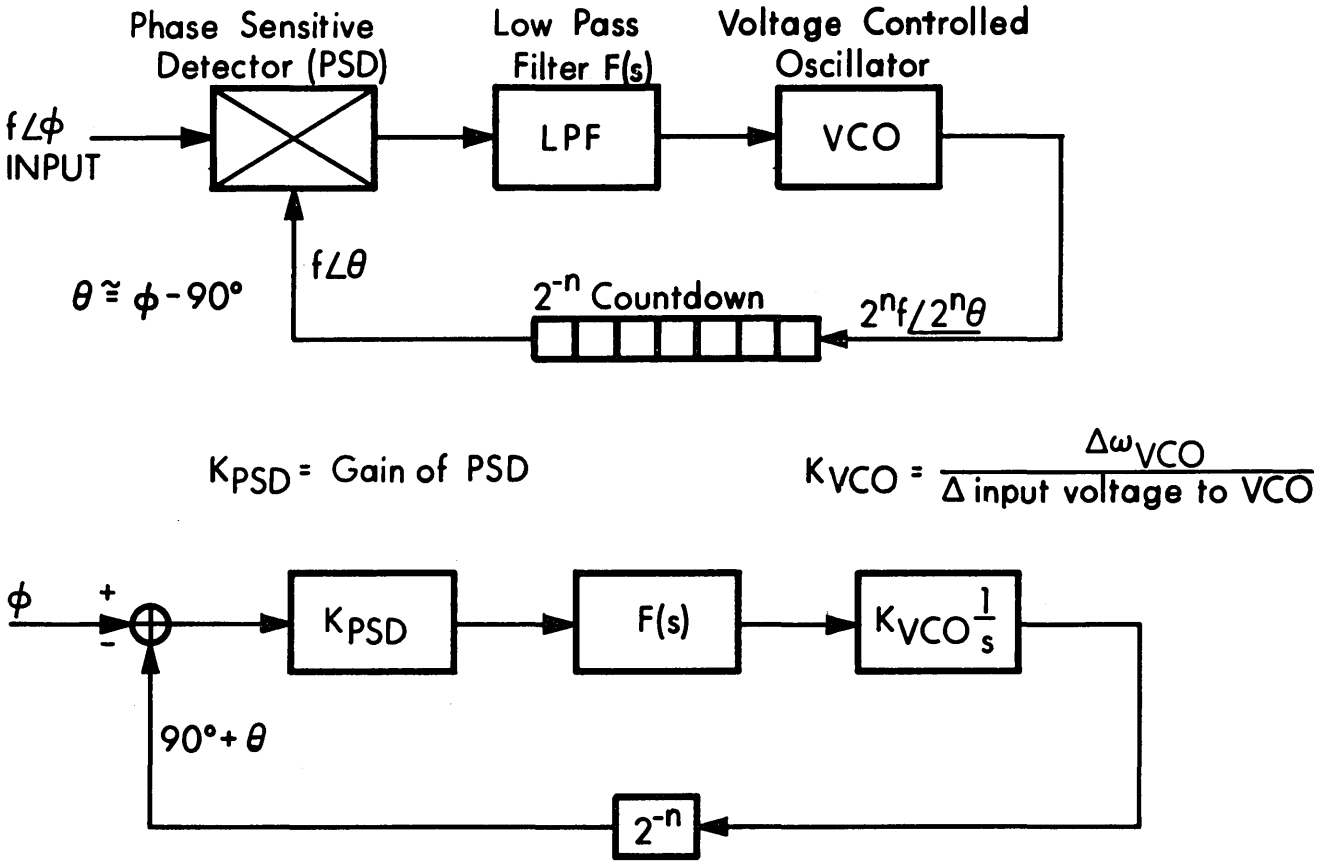


Figure 2—The phase-locked loop

respect to a reference clock waveform f∠0 is easily determined by the process diagrammed in Figure 3. At the instants when the clock waveform f∠0 changes states in the positive going direction a determination of the states of each of the countdown waveforms is made. Knowledge of whether f∠θ is up or down narrows the uncertainty in the angle to a 180° region; knowledge of whether 2f∠2θ is up or down further narrows the uncertainty in the angle to a 90° region; etc. This process, referred to hereafter as "strobing the countdown waveforms with the clock waveform,"* results in a unique synchronous binary encoding of the angle and is the heart of the new I/O. A functional implementation of the strobing technique to measure a resolver angle is diagrammed in Figure 4.

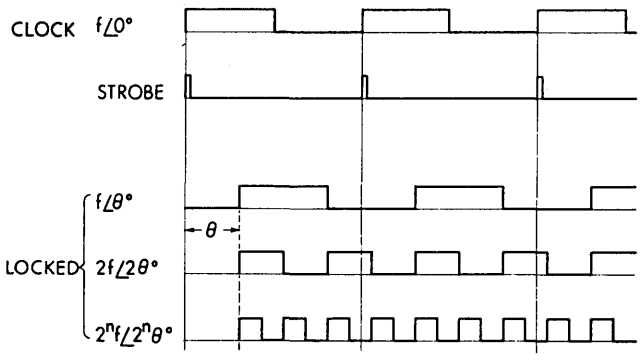


Figure 3—Strobing waveforms

The strobing process for obtaining binary encoding of phase angle can easily be extended to the case where there are two signals available corresponding to a common measurement variable, as from one-speed and m-speed resolvers on a common shaft. The one-speed signal $f_1\angle\phi$ and the m-speed signal $f_2\angle m\phi$ can, of course, be encoded separately, each with its own phase-locked loop strobed from a common clock. However, with this arrangement, as with any independent encoding scheme, unavoidable misalignment between the one-speed and m-speed data will cause,

*Note that the process could be modified to obtain angle encoding by having the signal f ∠θ from the phase-locked loop strobe the signals in the clock countdown, in which case the phase-locked loop would be acting as a filter and zero-crossing detector. However, with this arrangement, the precise times at which the data strobes occur would not be known *a priori*, and this uncertainty could be an important disadvantage when precise dynamic measurements are required.

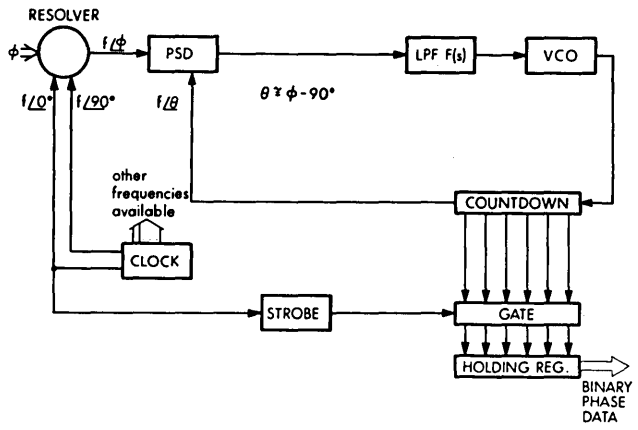


Figure 4—Strobing technique to obtain binary phase output

for certain angles, an inconsistency in the binary encoded data. An example of such data from 1-speed and 16-speed resolvers on a common shaft is shown in Figure 5. The inconsistency lies in the fact that the overlapping bits do not agree. In this case it is

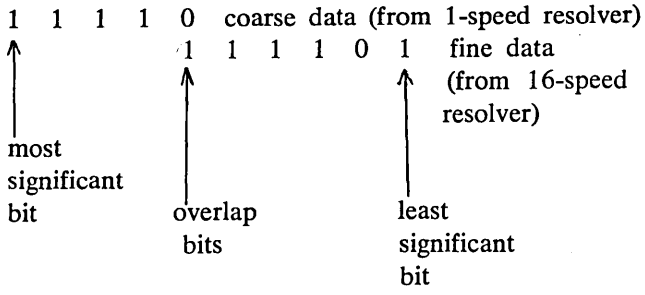


Figure 5—Data misalignment

necessary to establish the direction of phase misalignment and to add or substract a unit from the coarse word accordingly. However, with the new I/O a bit can be added to the coarse word merely by delaying the coarse-word strobe. Hence, an efficient and easily implemented technique for data alignment is as follows:

- Introduce a phase misalignment in the waveform f_1/θ from the countdown of the coarse-data loop such that the waveform corresponding to the least significant bit from that loop lags the waveform corresponding to the most significant bit from the fine-data loop by about 90°. Strobe the countdown of the coarse loop as usual (on the same clock signal as for the fine loop) if the overlap bits agree. Otherwise, delay the strobe of the countdown of the coarse loop until the overlap bits do agree. (Clearly, this is not the only possible method of data alignment. It bears similarity to a double-precision operation involving a carry bit.)

The introduction of the correction signal for misalignment angle is easily accomplished by adding a small fixed voltage to the signal at the output of the PSD of the coarse-data loop. Then the indicated strobing scheme results in whole-word data being obtained jointly from the one-speed and m-speed phase-modulated signals.

The elementary I/O as described has been constructed and operated satisfactorily to obtain 13 bits of whole-word data at a 1 KC synchronous strobing rate from a 1- and 16-speed shaft-angle resolver.

Comparison with conventional angle-measurement techniques

The elementary I/O is in essence a new and powerful technique for measuring and encoding phase angles of time waveforms. On this basis alone, without consideration of the additional processing and computing capabilities, the elementary I/O offers several advantages over conventional phase-measurement techniques.

Almost all phase measurement techniques require the conversion of the phase-modulated waveforms to rectangular or square waveforms which are in turn used directly for timing measurements. In the new I/O, as often in the field of radio telemetry, the conversion is made through the use of phase-locked loops. In conventional systems for measuring resolver angles the conversion to square waves is done through the use of zero-crossing detectors, perhaps preceded by band-pass filters. The phase-locked loop has a superior capability to discriminate against noise (thereby avoiding totally false readings due to multiple false triggering of level detectors) and, in addition, avoids errors that occur as a function of frequency shift in direct transmission through a band-pass filter. The new I/O broadens the area of application of phase-tracking techniques long known in the communication field.

Conventionally, the timing measurements on the square waves are performed by either of two basic methods: One of the methods is to use a linear analog phase detector to determine the phase of the zero-crossing detector output with respect to a clock waveform. The phase detector output must then be filtered and A/D converted to obtain a numerical representation of phase angle. This method relies strongly on the linearity of the phase detector, and results in significant dynamic errors due to the filter. The new I/O offers the advantage of utilizing a nulled phase detector, which need not be linear, and the advantage of greatly improved dynamic performance. It also has the advantage of relative ease and simplicity, (and therefore

economy) of A/D conversion. The other conventional method of processing the zero-crossing detector outputs is to use their rising and/or falling edges to start and stop a counter that is driven by the reference clock. The final counter outputs (at variable clock times) are the encoded phase angle data. This method is the open-loop counterpart to the strobing method of A/D conversion used in the new I/O. In the new I/O the counter is incorporated in the phase-locked feedback loop and numerical angle data is obtained *at known clock times* by the strobing process. Hence, the elementary I/O combines the signal-tracking capabilities of the phase-locked loop with the capability of synchronous angle encoding by the strobing process.

The new I/O has the additional advantage that it can easily be generalized to perform a wide range of simultaneous signal processing tasks that cannot be performed as effectively by conventional schemes.

Advantages and computing capabilities of the new I/O

Sensor as part of system memory

One important capability of the new I/O is the ability to provide synchronous whole-word data from a single or multi-speed source without the requirement of memory registers. After momentary interruptions in power sources, communication channels, etc., the current data word is completely restored. In a very real sense, the analog data source — e.g., the resolver — can function as part of the system memory and is an adjunct to the computer memory. Similarly, the I/O is akin to an addressing mechanism for the system.

Equally important is the capability to perform simply and rapidly a useful set of computing (data processing) functions, which will be enumerated presently. When a multiplicity of data sources are present these computations are performed simultaneously by the individual sections of the I/O, and the processed data can be addressed serially, randomly, or otherwise, by the GPC and/or by an off-line processor. In this manner the new I/O can take on a rather large real-time computing load which otherwise would have to be assigned to the GPC.

Filtering

One type of data processing function naturally performed by the I/O is filtering. Because the phase-locked loops are tracking filters with limited bandwidths, they serve not only to discriminate against electrical noise in the transmission path, but also to smooth the angle data itself. This smoothing function is particularly useful in the cases where the angle variations to be "smoothed out" are relatively rapidly varying. In those cases real-time smoothing by means of a GPC would require computations to be performed at a frequency more than twice the highest important frequency component in the power spectrum of the angle data. The new I/O performs the filtering operation automatically on the phase-modulated waveforms and thereby relieves the GPC of a rather large computing load that it is generally not designed to handle in the first place. The general purpose computer remains free to alter the filtering time constants according to either a programmed or adaptive control law by altering the parameters of the low-pass transfer function F(s).

Time-derivative data

Often, particularly when the data is being used in a control loop, it is desirable to obtain the time derivative of the variable being measured. When the data represent mechanical angle the GPC is often assigned to compute the derivative, a task which it cannot efficiently perform at high speed. The new I/O can provide the time-derivative (angular velocity) data directly in the form of a shift in the VCO frequency from its nominal value $2^n f$. Because both the frequency shift and the nominal frequency of the transmitted phase-modulated signal are magnified in the phase-locked loop by the factor 2^n (where n is the number of countdown stages), the determination of the shift can be made both accurately and rapidly. Many convenient and practical methods of measuring frequency shifts have been developed over the years for Doppler navigation, FM data transmission, etc. Any of these can be used to obtain the encoded time-derivative data. As an example, Figure 6 shows an analog frequency-difference detector followed by an A/D converter to provide encoded time-derivative data to the GPC. An alternate rate signal may be derived from the error signal to the VCO in the phase-locked loop and the choice is a matter of signal-level and hardware considerations.

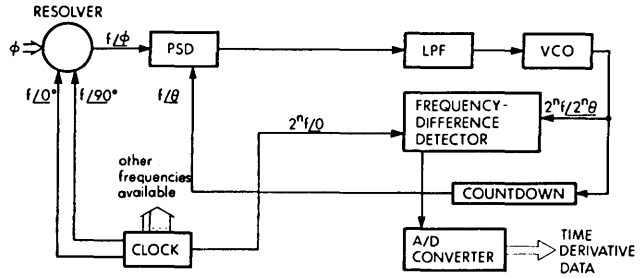


Figure 6—The generation of time-derivative data

Compensation of periodic instrument errors

Another type of computation that the new I/O can usefully perform is the removal of periodic instrument errors. Ideally the electrical phase angle of the transmitted data is linearly proportional to the quantity to be measured, e.g., shaft angle. Departures from linearity are objectionable and are often called "errors." However, the accuracy of the transducer may be considerably greater than its linearity, in which case it is desirable to remove the known non-linearity from the data numerically or otherwise. Although this is an extra and perhaps unwieldy task in real time for a GPC, the new I/O handles the task readily in the following manner: Suppose, for example, that the lowest-frequency signal f/θ in the phase-locked loop in Figure 2a is multiplied (an "exclusive-or" operation on square waves) by the signal f/θ from the clock. The average value of the product is a triangular function of the angle θ and, hence, of the angle Φ. If this product signal is added to the signal at the output of the PSD in the phase-locked loop, the output data angle θ will be displaced from the input data angle Φ b ythe triangular correction function (here assumed to be suitably small so that the effective gain of the PSD can be considered fixed). Similar correction functions can be generated with different periods and phase angles as a function of θ by using different frequencies and angle references in time, as indicated in Figure 7. By using a set of correction voltages generated in this manner, essentially any nonlinearity can be compensated for if it is a known periodic function of Φ. In this manner the binary angle (and angular rate) data is compensated for before delivery to the GPC. The latter remains free, for example, to oversee the correction process, perhaps to set adaptively the correction parameters, e.g., K_c and α in Figure 7, which can be stored temporarily or permanently in the I/O itself.

In Figure 7 the square wave $mf<m\theta$ is shown as

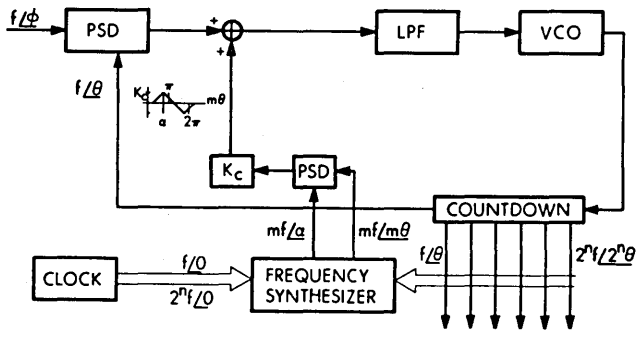


Figure 7—Correction for periodic instrument errors

an output from a frequency synthesizer. The function of the synthesizer in this case is to take the waveforms f/θ, $2f/2\theta$, $4f/4\theta$, . . . , $2^nf/2^n\theta$ from the phase-locked loop counter and generate square waves at any desired missing integral multiples of f, such as $3f/3\theta$. This can be accomplished ,with phase information preserved, by a novel technique, developed at MIT/IL, for frequency heterodyning.* The function accomplished by the heterodyner is diagrammed in Figure 8.

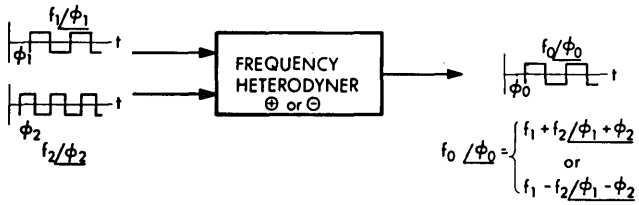


Figure 8—Fundamental element of frequency synthesizer

Correction of phase delay as a function of frequency

In some resolver applications, energy-storage mechanisms associated with the transmission path can introduce a phase shift into the carrier waveform as an approximately linear function of frequency. The new I/O can easily compensate for this phase shift, which can be considered as a dynamic phase error, by adding a small voltage from the frequency-difference detector to the voltage from the PSD as shown in Figure 9. This type of correction, not easily handled by the GPC, would be particularly valuable in applications where instantaneous angle data is desired from a resolver rotating at high angular velocities.

Redundancy reduction

In many applications, the behavior of the phase

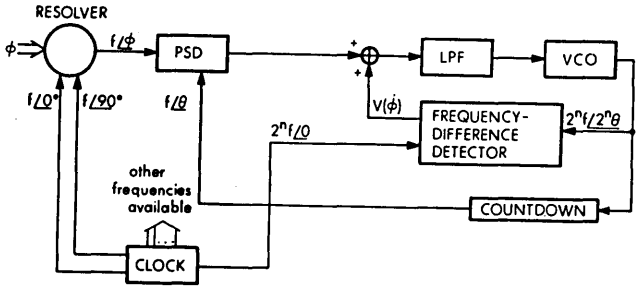


Figure 9—Phase correction as a linear function of frequency

*The technique, which is akin to single-sideband operations and results in a felxible frequency synthesizer,[4] was developed by Edmund Foster and Kenneth Fertig.

and frequency of the incoming waveform to the I/O is approximately known *a priori,* and the real data of interest is the departure of the actual data from the nominal data. Examples of this situation arise in a number of Doppler-type navigation systems when the approximate course of the vehicle is known from other sources of navigation information. In such cases it is useful to be able to reduce the computing load of the GPC by subtracting the expected data in real time from the actual data. This is easily accomplished in the new I/O.

Suppose that the expected Doppler shift is $K_1 f$ cps, where K_1 is some rational numbr less than unity. This can be removed from the data by using a strobe waveform

$$f\underline{/0} \oplus K_1 f\underline{/0} = (1 + K_1)\, f\underline{/0} \qquad (3)$$

where $\oplus$ indicates the heterodyning operation in Figure 8. If this strobe signal is used, and if the actual data is equal to the expected data, the strobed binary angle is a fixed number. Small deviations in the actual data from the expected data result in slowly varying binary angle data. The GPC in turn has to process only the slowly varying data, which contain the significant information, and can ignore the rapidly-varying unprocessed data containing redundant information on the already-known nominal path. By using the strobe waveform in place of the clock waveform as an input to the frequency-difference detector, the redundancy reduction of the time-derivative data is also effected.

The I/O can perform frequency correction also as a function of the frequency of the incoming signal. This is accomplished by letting the strobe waveform be

$$f\underline{/0} \oplus K_1 f\underline{/0} \oplus K_2 f\underline{/K_2\theta} \qquad (4)$$

as shown in Figure 10.

Figure 11 shows a functional diagram of the new I/O with provision included for the computing options mentioned thus far. By placing the phase-pre-

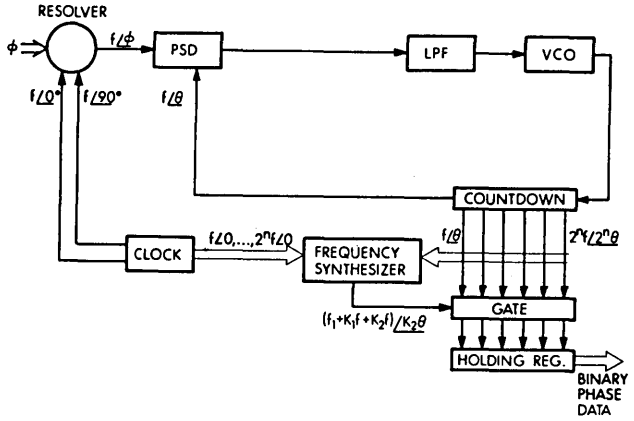


Figure 10—Method of correcting frequency as a function of frequency

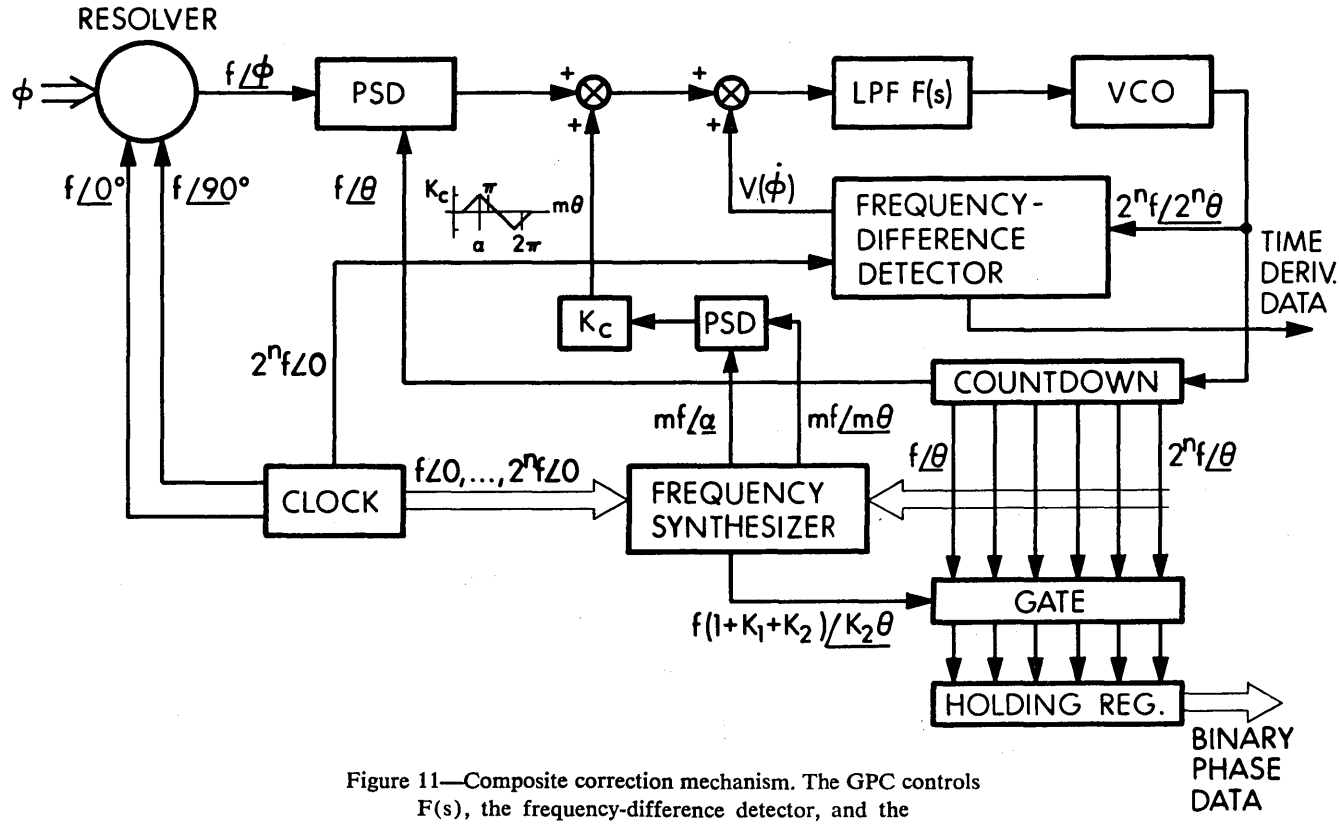


Figure 11—Composite correction mechanism. The GPC controls F(s), the frequency-difference detector, and the frequency synthesizer

serving frequency synthesizer under GPC control any schedule of expected data may, in principle, be used in the redundancy reduction process. Moreover, the GPC may be used to perform a monitoring function to determine the extent of redundancy and error reduction and to modify the synthesizer accordingly.

Other computing capabilities

The computing capabilities discussed in the foregoing are substantial but rather obvious once the basic operation of the new I/O is understood. Many other types of computations can be performed, and it is likely that those mentioned are only the beginning of a long list. Some of the other possibilities presently under investigation are briefly summarizd below:

(1) Ladder networks can be used in conjunction with the holding register to form linear and transcendental functions of the strobed angle data.
(2) A countdown in one phase-locked loop can be strobed with a signal from another phase-locked loop to obtain relative-angle data.
(3) Digital-differential-analyzer ideas can be incorporated to allow products of data words to be obtained.
(4) Generalized hybrid computation may be considered where the ladder networks of (1) are excited by voltages related to real signals. Depending upon where the feedback loop is closed with respect to input and ladder output, analog multiplication, division, etc., can be performed as shown in Figure 12. If the ladder is on the countdown, a continuous presentation of data on a modified carrier may be obtained. By using the multiplication and data-storage capabilities, correlation of data can be performed.
(5) By incorporating logic circuitry in the clock to allow generation of waveforms at independently controllable phase angles, GPC outputs may be phase encoded for subsequent processing by the new I/O. The I/O could present the results in digital, analog, or phase-encoded form.

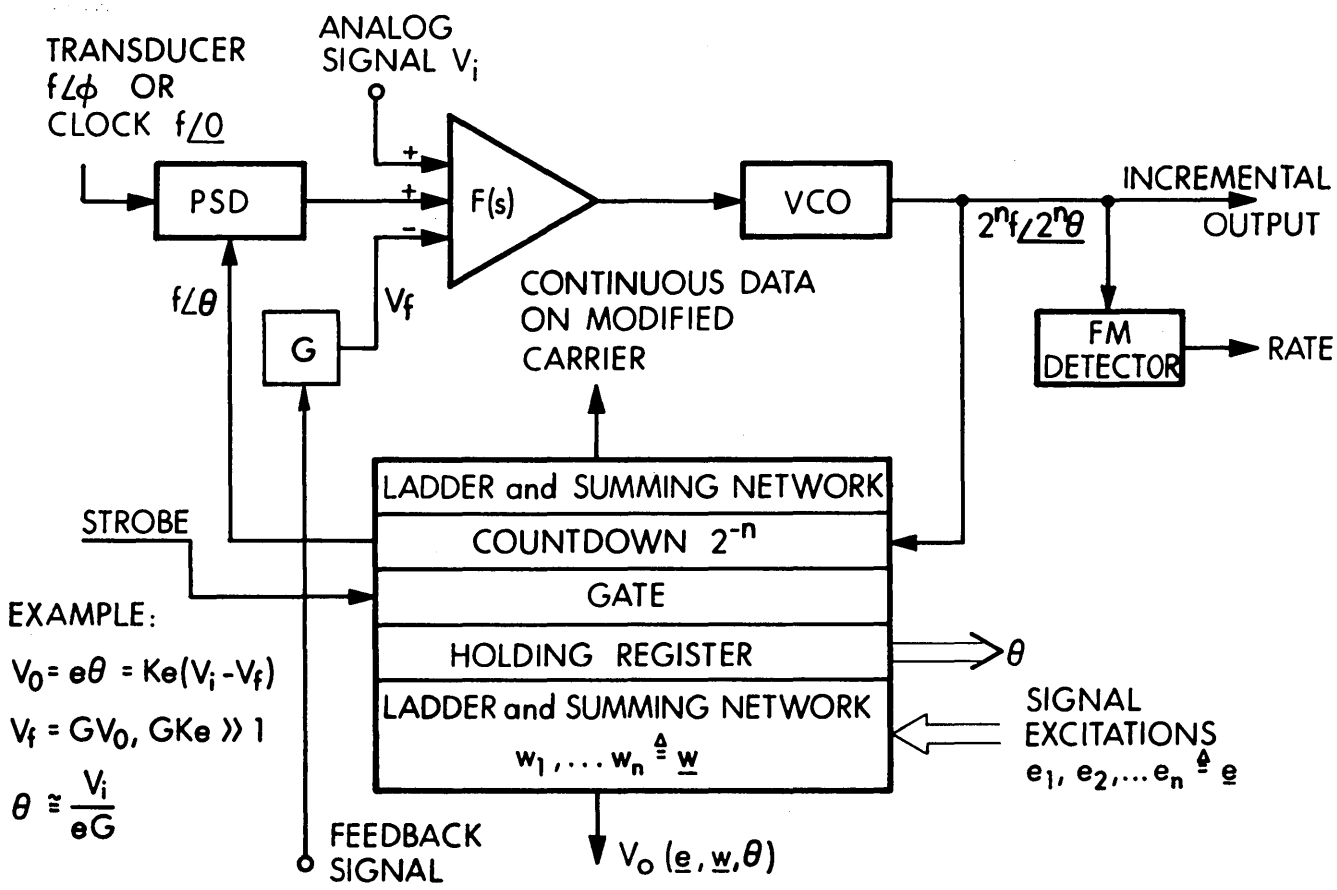


Figure 12—Hybrid computer element

Generalization: Instrument servo and new I/O

The properties of the new I/O result from the action of the VCO in the phase-locked loop. The properties of the instrument servo of prehistoric analog computer days were largely dependent on the motor which acted as an integrator in the sense that a voltage applied to the motor control winding resulted in an angular velocity, the integral of which is angle. Similarly, the voltage input to a VCO controls frequency whose integral is phase. In the old servo, the motor shaft could be loaded by tachometers, resolvers, synchros, potentiometers, etc. In our phase-locked loop, the VCO and counter may be loaded by FM detectors (note that the loop error signal is the Doppler shift), ladders, etc., as shown in Figure 12. Clearly, the similarity between the phase-locked loop and the instrument servo is being stressed. From this similarity, all of the computing capability obtained previously with the instrument servo may be obtained from the electronic servo — the phase-locked loop.

As a result of the preceding discussion, a further generalization may be made which is incidental to the discussion of I/O. The phase-locked loop is really an accurate operational amplifier and the technique may be used to make a family of analog computing elements, servos, etc.

Hardware implications

The circuit elements which are used to make a phase-locked loop lend themselves to microminiaturization. The developments in integrated circuits and large scale integrated arrays make hadware I/O improvements attractive.

Further, examination of existing I/O's for several real-time computer/control systems shows that the system designers, in order to alleviate the computer data-processing load, have placed in the I/O adders, forward-backward dual-rank counters, A/D converters, etc. The phase-locked loop, utilizing a forward-only counter, etc., appears to lend itself to a building block concept which represents no increase in hardware but rather a regrouping and a considerable increase in flexibility.

CONCLUSION

A simple high-speed I/O with real-time computing capability has been described. The I/O is particularly well suited for performing high-speed calibration and redundancy reduction on data received as phase modulation on carrier waveforms, and for presenting the processed data in a convenient whole-word binary format for further processing by a general purpose computer. The I/O is non-incremental in nature and, hence, is completely self-restoring after momentary equipment interruptions and can be adaptively manipulated under general purpose computer control.

The new I/O is a step in the direction of freeing a general purpose computer in a real-time measurement and/or control application from high-speed but routine data-reduction tasks (for which it is not well suited) to perform computation, sophisticated monitoring and adaptive adjustment tasks (for which it is particularly well suited). Although the examples have been about inputs to the general purpose computer from continuous sources, the techniques described would apply to the generation of outputs from the general purpose computer which commands or controls some element of a system.

The concepts of I/O have been generalized to include the general problem of analog computation and simulation and to lead naturally to hybrid computation techniques.

ACKNOWLEDGMENT

This report was prepared under DSR Project 52-28611-22S sponsored by the Ballistic Systems Division of the Air Force Systems Command through Contract F04694-67-C-0028 with the Instrumentation Laboratory of Massachusetts Institute of Technology in Cambridge, Massachusetts.

REFERENCES

1 L J QUAGLIATA
A sampling technique for the transmission and recovery of phase-encoded information
Master of Science Thesis Massachusetts Institute of Technology Cambridge Mass Instrumentation Laboratory Report T-428 June 1965

2 R C TAUSWORTHE
Theory and practical design of phase-locked receivers
Vol 1 Jet Propulsion Laboratory California Institute of Technology Technical Report No. 32-819 15 February 1966

3 F M GARDNER
Phaselock techniques
John Wiley & Sons Inc New York 1966

4 E FOSTER
A circuit to perform frequency addition of square waves
Master of Science Thesis Massachusetts Institute of Technology Cambridge Mass Instrumentation Laboratory Report T-475 January 1967

On designing generalized file records for management information systems

by FRANK H. BENNER
Bell Telephone Company of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

The centrality of the files in a MIS (Management Information System), or in any other large scale computer application, has been traditionally considered as self-evident. The new problems of operating systems and the languages for use in the 3rd generation computers, plus the relatively unknown world of integrated data communications may tend to remove the files from the limelight. This should be avoided. Most business applications will continue to be "file bound," or I/O limited, even with random access to the data base.

A file reorganization to correct design deficiencies can be catastrophic in its impact on programs that were "on-the-shelf." The majority of the system hardware cost is probably charged to file devices. The continuing requirements for an optimum design for the complete system is recognized and honored. It is suggested that the search for this elusive design solution should begin in the files area. Here the payoff is handsome for success, and the fiscal and operational deficit is unrelenting for failure.

In a changing business environment, the mortality of a special-purpose file organization is expected to be high. The prudent approach for the designer is to make no a priori assumptions about the way in which the data base will be used, new uses will arise; or the limit of the required descriptors, new data will be added. Both R. V. Head[1] and W. H. Desmonde,[2] while describing different real-time systems, are strong in their emphasis on file organization as a limiting factor on throughput and cost, which requires rigorous optimizing techniques. A new approach to file design seems required which takes advantage of the "naturalness" that may exist in families of information. Paraphrasing Alexander,[3] the difficulties attended upon third generation computer applications are much more subtle and complex than in the past. As in the usual case, the temptation to fall back on some arbitrarily chosen order or design technique is almost overwhelming. But, if we continue to apply essentially punched card or tape file design techniques to the direct access file problem, we will live with essentially an unsolved problem and pay for the designers comfort.

Design criteria

This paper reports on a method for analyzing the logical record requirements and designing the physical records to be housed on direct access storage devices. This methodology has been followed in the Pennsylvania Company to design the file system for BIS (Business Information System), a specific term used in the Bell System to identify an indigenous MIS. This system is characterized by large direct access files, a variety of real-time activity to the files with "background" work during the business day, and intensive and periodic batch processing of the files on non-prime time. These characteristics are, in many ways, similar to the corporate MIS for many businesses.

A MIS usually consists of two operational categories of programs; real-time and batch. The file system to support these processes must at least consider;

1. Security of information.
2. Recovery of system operation after failure, and restoration of data when mutilated.
3. Key and Addressing schemes that will result in densely populated storage.
4. Need for additions and changes to the data base to meet changing requirements.
5. Performance requirements of all types of activity.
6. Reduction of wasted mass storage, consistent with performance requirements.
7. Coordination of the file system with the programming system.

Of the above, security, recovery, restoration and addressing do not lend themselves to a universal treatment. The speed requirements for recovery of operation and

correction of mutilated data is not the same for all systems. Some can tolerate times in terms of days; others demand action in minutes. Economics is also a factor in designing for system assurance. Duplexed files are expensive, but not always required. Each application has its own unique security problems and scheme for identifying records. The requirement for an optimum file record is present in all applications however.

The record designer must be able to qualitatively describe this optimum record before attempting his work. When this record is designed, his search is ended.

The optimum record will:

1. Utilize the greatest amount of the space allocated at the home address of the storage device.
2. Permit selection of data from within the record.
3. Be structured so that data fields can be added, eliminated, and rearranged.
4. Identify records by using a logical description, in addition to absolute identification.
5. Be as short as possible and yet, in the selected length, supply the greatest amount of file data to the programs in one "seek."

Logical record considerations

Most business applications will be dealing with a logical record in the data base. In an Inventory Application, the logical record may be the set of all the subsets whose elements contain information primarily about a particular Piece Part. In an Order and Billing Application, the logical record is usually a Customer Account. With a total systems approach as used in a MIS, those applications which were previously separate bounded systems, now are subsumed into the MIS or are subsystems. If we were to arbitrarily retain the demarcation between say, an Inventory System and an Order and Billing System in an MIS environment, there would be unnecessary redundancy of data and a high likelihood of subjectively favoring one of the subsystems of the MIS at the expense of the others.

The amount of information making up a logical record is usually not homogeneous throughout the universe of the records. Some of our inventory items or our customers are more active or larger than others. Thus, the logical record size will most likely vary. We could make each physical record the size required to accommodate information in our largest logical record. This would give excellent performance, but low utilization of storage would result. We could make our record length cater to the "average." This would give us good storage utilization. Since it is usually the larger and complex entities that are most active, "average" record length results in lower system performance. A disciplined approach to record design is required to properly weigh all factors.

Attributes of record components

Analysis of the natural components of a logical record is aided by the diagram, Logical Record Components, Figure 1. We see that a Logical Record is the superset of functionally related Main and Auxiliary Records uniquely identified with a particular entity. The subsets of the Main and Auxiliary Records are the Data Segment(s) and one Record Control Segment. A Data Segment is a collection of fields that cover a particular aspect of the Logical Record. Then, the elements of the Data Segment are Data Fields, with possible intersection shown. Each Logical Record has one Main Physical Record and zero or more Auxiliary Physical Records, depending upon the amount and kind of data making up the record.

There has been an increased interest in memory systems where information is stored or retrieved on the basis of content. This is in contrast with using specific addresses to reference locations containing data that must be examined. This rationale is generally referred to as being "associative" in orientation.[4] Since about 1961 the term "associative criterion" has been used to define the list of descriptors for the desired group of data or record. In this context an item for/of entry in an associative memory is described by as many characteristics as are required. The object record is located or identified at the intersection of all these descriptors. Using the associative philosophy of the memory system designer, it is a short step into a new logical organization for file records.[5] Each record can still be specifically identified, but it also may be understood from the associative logical criterion. Each set of descriptors then describes an object, but the object described is not always congruent with only one specific record. This idea permits the accommodation of more than one logically consistent file in the same physical file structure. The value of such an organization is its generality of logical flexibility.

Fitting the physical records into such a file requires an examination of the record elements. In a direct access record, the basic building block is the data field. Close ties between fields are reflected by the creation of data segments. We must know specific information about the data fields so we can determine the particular components of an optimum main record These attributes relate to:

1. How often the field even exists to contain significant data?
2. When it does exist, how long is the field?

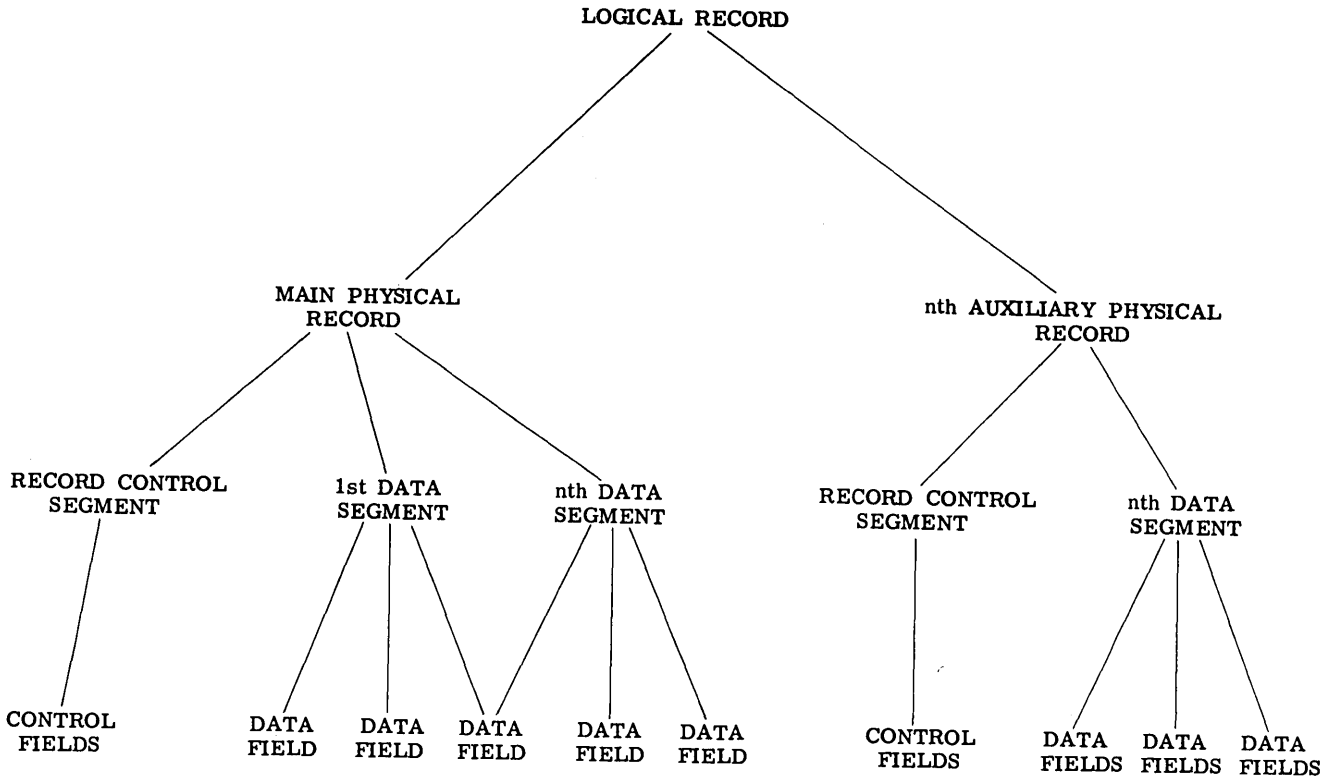


Figure 1—Logical record components

3. How often is the field required when the Logical Record is processed?
4. When it is required for use, what other fields are also needed?

The information pertaining to how often a field significantly exists aids in attaining efficient use of storage. The length of a field and its variance help determine the use of storage, and how often the system will be required to access "overflow" data. The way in which, and how often, the data is used in servicing transactions or processing requirements will permit a design that lets each application "stand up and be counted" on its true need.

These facts about a mythical record are shown on Data Segment and Field Attributes, Figure 2. Each column on the chart contains specific information about the data field or data segment that appears on a particular row. Note that Segments A and D have no component fields shown. This is because the fields are a tight family, (exist with practically the same probability and level of activity) so they are considered as being one large field. The column headed Prob. refers to the probability that the field will contain significant data in the Logical Record.

$$P_i = \frac{E_i,}{N} \quad P_i \leqq 1.00 \qquad (1)$$

where: E is number of Logical Records in which i^{th} field $\neq$ O,

and: N is number of Logical Records in the data set

The Mode column refers to the characteristics of the field in terms of being fixed (F) in length or variable (V). In fixed length cases, only the Min. column is used. The details of how a length varies are required only for variable length fields or segments. This is done by expressing the field length and frequency as a series of coordinates in the columns under Data Length headed Pt. 1, Pt. 2, Pt. n. In these columns, L and f (L) are shown for the various points. The Min column pertains to the shortest length the field was found to have.

Activity on the system is expressed on Figure 2 in terms of the frequency of need in a given time period for the particular fields and segments. The Real-Time column pertains to the transactions to be conducted in real-time. In the illustration, two different kinds of real-time transactions have been shown.

Type 1 requires Segment A, Segment B Field 1, and Segment C, Fields 1 & 2

Type 2 requires Segment A, Segment C Field 1, and Segment D

The Batch column refers to the requirements for

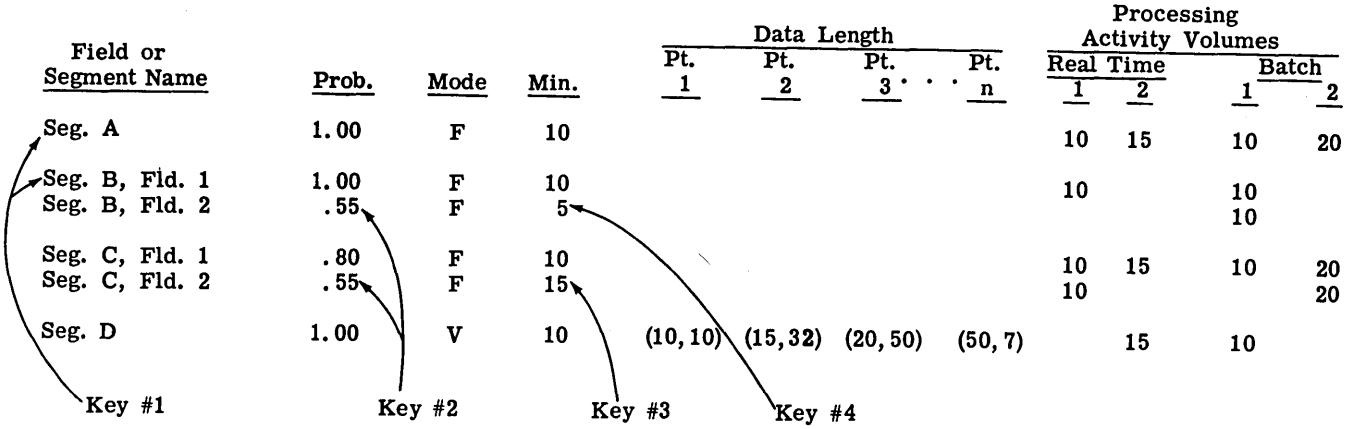


Figure 2—Data segment and field attributes

information to meet these similarly appropriate needs. The volumes shown must be taken for the same period of time. Total activity to a Segment or Field is simply:

$$A_t = E \sum_{i=1}^{n} RT_i + \sum_{j=1}^{n} BTCH_j \qquad (2)$$

where: i is a real-time transaction
j is a batch transaction
E is $\geqq 1$

and: A_t is Total Weighted Activity

The strategy to be employed later will involve certain computations that use the length, probability, and the activity for each of the data segments. Most of these file systems must serve both real-time and batch processes.

If the file is to have no particular orientation, all of values L, P, and A can be used as they are found. If there is to be an orientation toward the real-time requirements, the records should be so designed. The factor E in the above equation (2) is used to effect this orientation. It is a number, proportional to the relative importance of the real-time activity. Arriving at this value is similar to assigning the priority or the transaction limits for the use of the control package or operating system. For instance, the control program might be arranged to permit a limit of say three or four real-time transactions to be served and then one background transaction. This same limit can be effectively used as E.

From an examination of raw activity, shown on Data Segment and Field Attributes, Figure 2, and, if we wish to emphasize the influence of our Real-Time activity by E = 2 as discussed earlier, equation (2), we have weighted values:

Segment or Field		Weighted Real-Time	Batch	A_t*
Segment A		50	30	80
Segment B,	Field 1	20	10	30
	Field 2	—	10	10
Segment C,	Field 1	50	30	80
	Field 2	20	20	40
Segment D		30	10	40

*Total Activity

By using this Total Weighted Activity, along with the related probability and the length we will be able to evaluate the relative importance of a field or segment.

Utilization of mass storage

The designer faces an implied constraint in the form of the economic use of storage. This is evidenced in his reluctance to provide space in the record for fields that occur infrequently. In tape records, this problem could frequently be taken care of by the placement of these fields where "zero suppression" could keep the tape record shorter if the field contained no significant data. In direct access devices there are as yet no such aids, so the decision cannot be so easily determined. The designer must have an objective in terms of storage utilization. Storage utilization is simply the probability that a position in storage will contain significant data.

Fixed length data fields

From the data on Figure 2, the designer has no problem with fixed length Segment A and Field 1 of Segment B, since there is a P equal to 1, Key #1. This tells him that they always significantly exist. The P for Field 2 Segment B, and Field 2 Segment C, Key #2, with P equal to .55, present a dilemma. If his objec-

tive is a utilization say of .70, the designer is tempted not to reserve space for these fields. This is fallacious if we consider only the lengths of the fields, since if a field might not exist, the problem programs must be able to test a control field associated with the object field to see if it is present.

Pan[6] in his pioneering work provided useful formulae for applying linear programming techniques to help solve the record design problem. In this particular example, after Pan, we can test the fixed length fields Field 2 Segment B, Field 2 Segment C, Key #2, and determine if we should always provide space in our physical record for these fields. To do this, the term Weighted Storage Utilization is used which is essentially the probability for the field, weighted by the ratio of a control length to the data length of the field.

$$W_i = \frac{L_i + C}{L_i}(P_i) \qquad (3)$$

where: L_i is the data length of the ith field
P_i is the probability of the ith field
C is the length of the control field used in the programming system.

If the programming system requires 3 digits for control to provide field identification and length, we have for Field 2 Segment C, on Figure 2:

$$W_i = \frac{L_i + C}{L_i}(P_i)$$

$$= \frac{15 + 3}{15}(.55)$$

$$= .66$$

where: L = 15 per Key #3, Figure 2.
For Field 2 Segment B, on Figure 2:

$$W_i = \frac{5 + 3}{5}(.55)$$

$$= .88$$

where: L_i = 5 per Key #4, Figure 2

Assume the rule, "Always allot space for data in the physical record when the Weighted Storage Utilization of the field equals or exceeds the design objective." Recalling our objective of .70, space will always be provided for Field 2 Segment B, $W_i = .88$; but none will be reserved for Field 2 Segment C, $W_i = .55$. If the application programs are "process" limited, borderline cases may be judged accordingly. This is due to some additional processing that is required to locate data using a control field, rather than by direct use of a "label" that designates specific positions in memory.

Variable length data

The problem with variable length fields, in terms of utilization of storage, is more complex. Here, in addition to treating the probability of the field, we also face the problem of deciding on a data length to provide on the first "read" of the record containing the field. Most methods of organizing records for direct access devices use a "chaining" or "overflow" address when data exceeds the space initially allotted. This "chaining address" field, incidentally, should be treated as any other fixed length field since it significantly exists only when overflow exists. The use of the additional detail on variable length fields is fairly obvious. It wlll be used to produce a probability function for each particular field. The skew present in a frequency distribution of the data has shown this essentially social data transforms, with good fit, into a form of the Gompertz Curve.[7] This curve fits well into a probability function where we encounter the probability showing an increasing ratio of decline as field length increases, but the ratio is not changing by either a constant amount or percentage. While this curve:

$$Y = ka^{b^x} \qquad (4)$$

is mathematically attractive, it cannot easily be managed for solution and integration in a digital computer. Raw data is used directly for this purpose.

The need for integration is to develop the storage utilization for variable length data. Recall that this is the probability that a position of allocated storage will contain significant data, or the ratio of the allotted area to the occupied area. For variable length fields:

$$W_i = \frac{\sum_{i=0}^{n} f(L_i)dL + C}{LA_i}(P_i) \qquad (5)$$

where: i is a particular variable length field,
LA is length of allotted area, and,
C is the length of chaining address data.

Morse[8] treated a similar problem in computing Mean Service Time by using Taylor's Theorem. While our problem is not one of queues, it can be treated in a parallel manner. Figure 3 is a graph of the Probability Function of a Variable Length Field. First, consider the probability that the field length, for any record in which the field appears, is greater than length L. This is the difference between the probability that the field will be of length (L) and (L + dL).

$$P(L) - P(L+dL) = P(L) - P(L) - \frac{dP}{dL}dL = p(L)dL \qquad (6)$$

where: $p(L) = -\frac{dP}{dL}$ or $P(L) = \int_0^\infty p(L)dL$

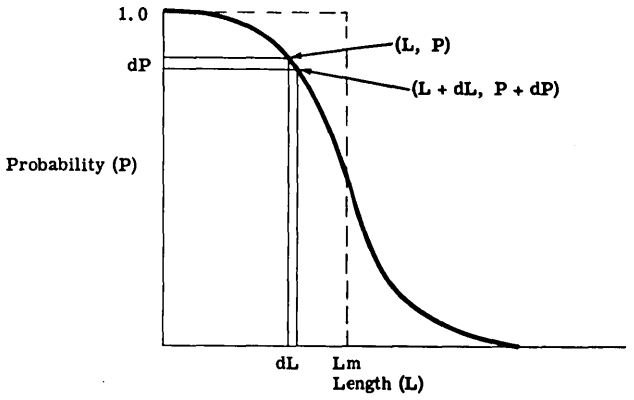


Figure 3—Probability function of a variable length field

This quantity p(L) is the probability density that a field is of length L. This is a rate or change in probability per length. Then p(L) is a measure of the mean rate at which Fields are meeting the length (L). The average length of the fields is:

$$L_m = \int_0^\infty L\,p(L)dL$$

$$= -\left[L\,P(L) \right]_0^\infty + \int_0^\infty P(L)dL$$

$$= \int_0^\infty P(L)dL \qquad (7)$$

where: L_m is the average length.

Discarding the quantity in the square brackets is justified since it is zero when (L) equals zero; and when $(L)\rightarrow\infty$, P(L) has already reached zero for all practical purposes.

Using the Trapezoidal Rule for approximation of the definite integral of the Probability Density in Figure 4, Functions for Segment D:

$$A \approx \int_{X_0}^{X_n} f(x)\,dx \qquad (8)$$

$$\approx \frac{\Delta x}{2}(Y_0 + 2Y_1 + 2Y_2 + 2Y_{n-1} + Y_n)$$

$$\approx \frac{5}{2}(1 + 2 + 2 + 1.88 + 1.52 + .96 + .54 + .38 + .26 + .16 + .04)$$

$$\approx \frac{5}{2}(10.74)$$

$$\approx 26.85 \approx 27$$

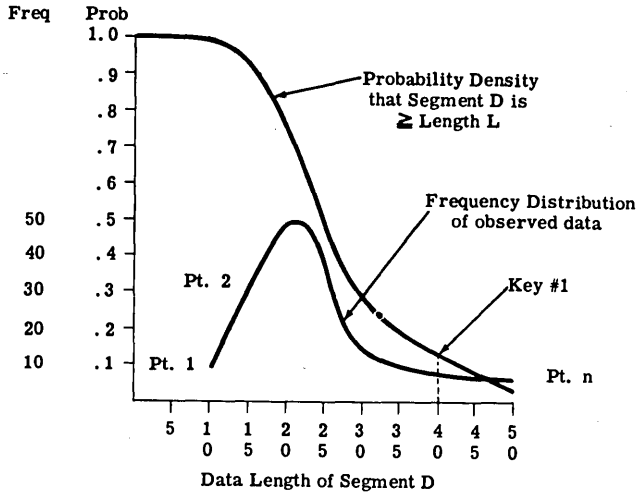


Figure 4—Functions for segment D

Applying this result to equation (5) for Segment D, and assuming "C" equals 3 digits,

$$W_i = \frac{\sum_{i=0}^{n} f(L_i)\,dL + C}{LA_i}(P_i) \qquad (9)$$

$$= \frac{27 + 3}{50}(1.00)$$

$$= .60$$

This would be true if LA_i is set equal to the maximum length the field was found to have. Therefore, the length of the allotted space should be reduced until W_i equals the objective, if possible. Through an iterative process we arrive at:

$$W_i = \frac{26 + 3}{40}(1.00) \qquad (10)$$

$$W_i = .72$$

So that if we allot 40 characters to Segment D, we will have a Storage Utilization of .72, the field will always appear in the physical record. From Figure 4, Key #1, the probability that Segment D≧40 characters is .12 or; about 12% of the times Segment D is required, an additional read would be required to secure the excess data. The number of device read commands that must be given to completely satisfy the need for specific data is an important factor on through-put. Therefore, it is necessary to keep file action to a minimum. The ability of a file record to meet the needs for data is expressed as "Performance," and is calculated as follows:

$$C_r = \{S_1, S_2, \text{----}S_n\} \qquad (11)$$

$$P(C_r) = \frac{\sum_{i=1}^{n} H_i N_i}{\sum_{i=i}^{n} N_i} \qquad (12)$$

Where: S_i is a Data Segment provided for in the combination C_r being rated for Performance, and, $P(C_r)$ is the Performance of C_r.

Where: i is the number of different segment combination required by transactions.
N is the number of times a particular combination is requested by processing needs.

and: H_i is a "hit" factor
$H_i = 1$ if all required segments are found with C_r
$H_i = 0$ if one or more of the required segments are not with C_r

Example:

$C = \{a, b, c, d, e\}$ is the combination being rated for the main physical record.

The different combinations requested:

Combination	H_i	Processing Volumes
$1 = \{a, b, c,\}$	$H_1 = 1$	$1 = 50$
$2 = \{a, c, e,\}$	$H_2 = 1$	$2 = 40$
$3 = \{a, b, f,\}$	$H_3 = 0$	$3 = 10$

then:

$$P(C_r) = \frac{\sum_{i=1}^{n} H_i N_i}{\sum_{i=1}^{n} N_i} \qquad (13)$$

$$= \frac{(1).(50) + (1).(40) + (0).(10)}{50 + 40 + 10}$$

$$= \frac{90}{100} = .9$$

Storage Utilization is a similar rating that measures the expectation that a character position of storage will be occupied by useful intelligence from the data base.

$$SU = \frac{\sum_{i=1}^{n} L_i W_i}{\sum_{i=1}^{n} L_i} \qquad (14)$$

where:

L_i is Segment length
W_i is Weighted Storage Utilization for the segment

Record design strategy

The designer will naturally attempt to produce alternate main record designs that will differ in:

1. Length
2. Performance
3. Utilization of Storage

There are a number of techniques that efficiently utilize available storage addresses.[9,10] In this strategy, we attempt to make the best use of the capacity available at the addresses. The combination of segments chosen for the Main Physical Record will determine the number of times that required data is not initially available to the system activity. In this procedure, the designer first expresses these factors:

1. The length of the control fields to be used by the Programming System for Variable Format Fields (Figure 5, Memory Map of Data Segment, Fields I_i and L_i).
2. The length of the chaining fields when used for variable length data.
3. The factor "E", as discussed earlier, gives emphasis to real-time activity in the design.

The basic strategy is shown on the flow chart on Figure 6, Design Procedure. When the Weighted Storage Utilization is low, most fields will be made fixed in format, and variable length fields will seldom require a second access to overflow. The record will be long and the Storage Utilization will be low. For each value of Weighted Storage Utilization there will be some length where the Performance is acceptable. The designer then plots these Performance and Storage Utilization results for each of the feasible lengths. The optimum record length is selected for implementation; the contents of the record is the combination of segments used to develop the selected results. On Figure 7, Graph of Performance and Storage Utilization, is shown the plot of Performance (solid) and Storage Utilization (dotted) that were generated from information for a portion of the BIS File System.

As record length increases, Performance increases since more segments are present in the record. Conversely, Storage Utilization decays since data with lower P are being included. There is a minimum performance that the selected record length must provide. The decision on exceeding that length is a tradeoff between the increase in performance that will be enjoyed, versus the decrease in the utilization of storage that will be encountered. For instance, on Figure 7, if

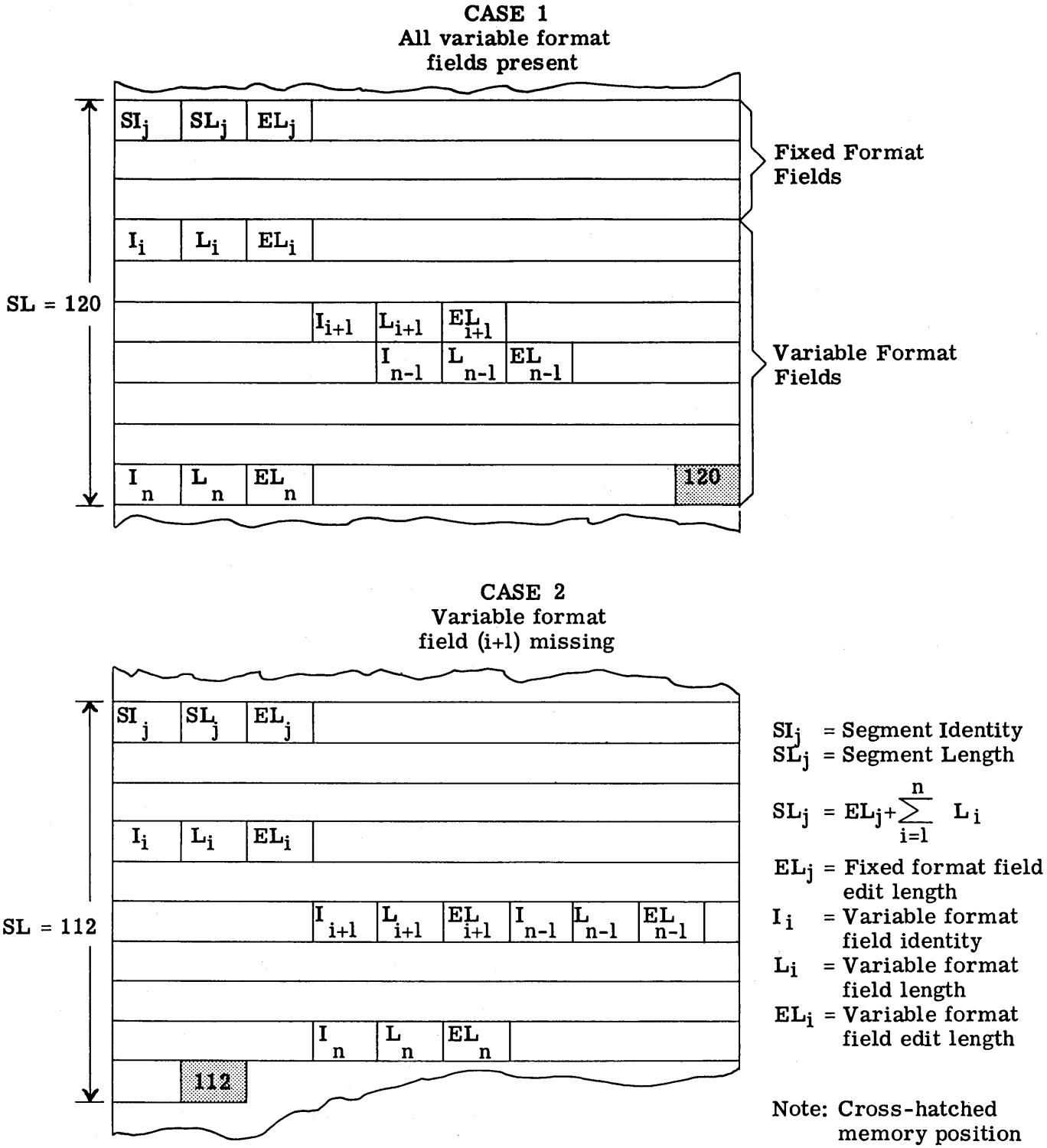


Figure 5—Memory map of data segment

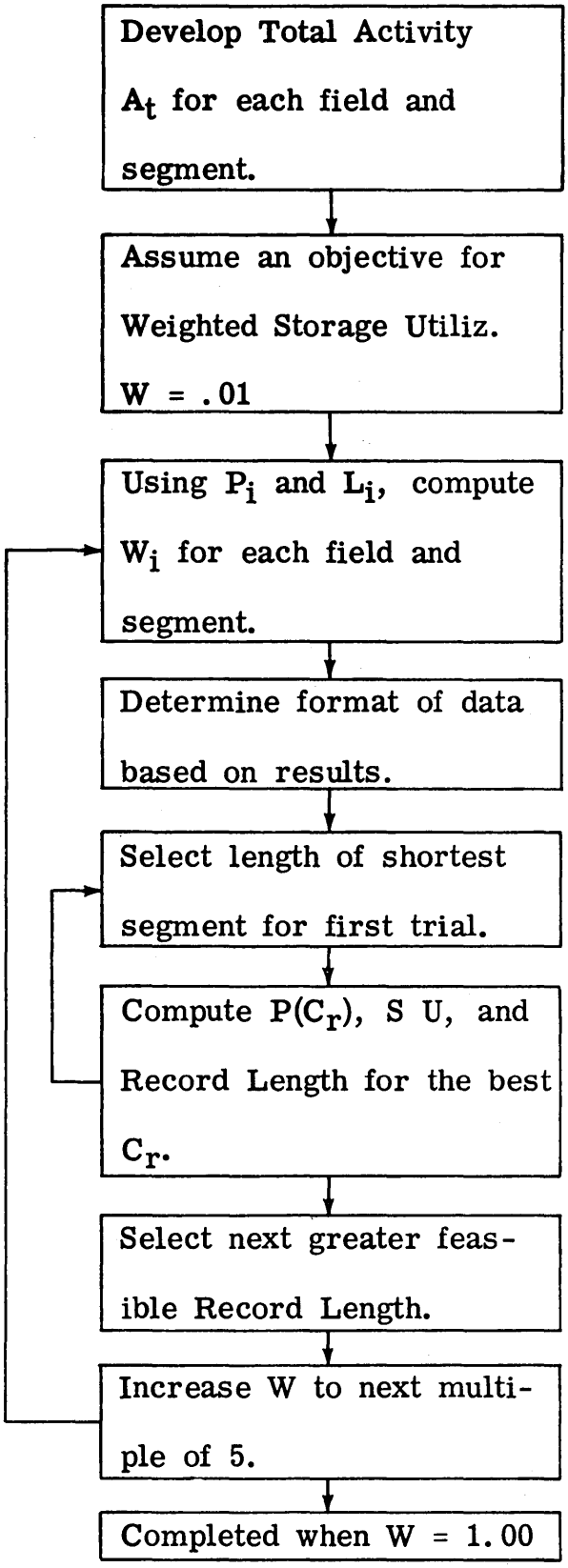


Figure 6—Design procedure

a minimum Performance of .90 is required, the length of such a record will be 600 characters. If the next feasible length is 700 characters, the decision to use this length is based on the absolute change in Performance and in Storage Utilization. A useful rule is to subject the change, or Gain ratio to a slope test:

$$G = \frac{\Delta P}{\Delta SU} = \frac{.02}{.05} = .4 \text{ If } G \geqq 1, \text{ take next length} \quad (15)$$

There are times when certain solutions might be ruled out due to address breaks, device constraints, and economics. This action should only be taken after the theoretical solution is found. These techniques may be difficult to use without some computer aids. Most designers will have a linear programming package available to them, or can have one written. Many of the ideas in this design method appear in a program[11] available from the IBM Corp and used in the Pennsylvania Company. Certain restrictions and specific understandings are in force when it is used.

After the contents of the Main Physical Record has been decided, the rest of the logical record will be in the Auxiliary Record(s). These records will contain seldom used data and the excess of the variable length data from the main record.

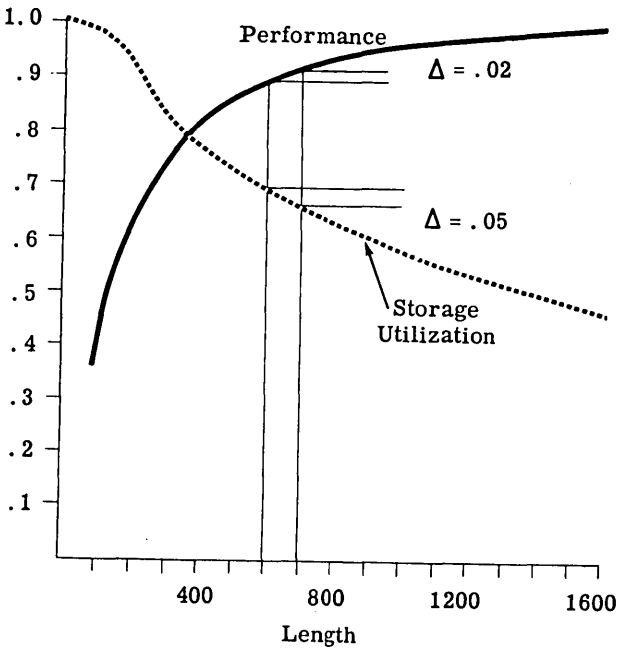


Figure 7—Performance and storage utilization

Programming considerations

Until now there has been emphasis on storage utilization, throughput and efficiency. However, it is necessary to place the resulting record in the context of the programming system. A fairly representative selection of storage devices[12,13,14] show a variety of capacities and speeds available. Capacity and high cost has been softened as a limit. On Figure 5 is shown how a segment will appear in main memory after it has been read from the device. Each segment must carry its identity (SI_j), or, type of data it contains. The total length of the segment (SL_j) must be shown for programming housekeeping. The segment length is a function of the length of the fields present.

$$SL_j = EL_j + \sum_{i=1}^{n} EL_i \quad (16)$$

The field designated EL is used to show field length and certain edit information. Note that Case 1 is longer than Case 2 for the same segment, on different records. It will also show the mode of the data:

Packed Decimal—Unsigned
Packed Decimal—Signed
Binary
Eight Bit etc.

All this information will be used by the File Interface Program to be discussed later.

Recall that each physical record will contain a Record Control Segment and one or more data segments. Within the Record Control Segment, Figure 8, Memory Map of Physical Record are fields to show the length (RCL) of this segment, the total length (RL) of the record data on the track, and the address of Auxiliary Physical Record(s). Figure 8 shows the record and some of the fields mentioned. It can be seen that:

$$RL = RCL + \sum_{j=1}^{n} SL_j \quad (17)$$

All Record Control Segments in the file system are managed the same way. After a record is read into memory, the start of the Record Control Segment will always be known. By looking at field N, the programming system can initialize to handle the number of segments possible in the subect file record. This is an aid to setting up "looping limits." Any segment can be located in the input area by referencing the positionally significant segment bit (SB_j) that pertains to the desired segment. This bit has two states:

Set = the segment is present in the record.
Reset = the segment is not in the record.

When the bit is found "set," reference is made to the associated Displacement Field (D_j) for the segment. This field contains the number of positions away from the start of the record where the segment is found. By consulting control fields in the segment any data can be secured.

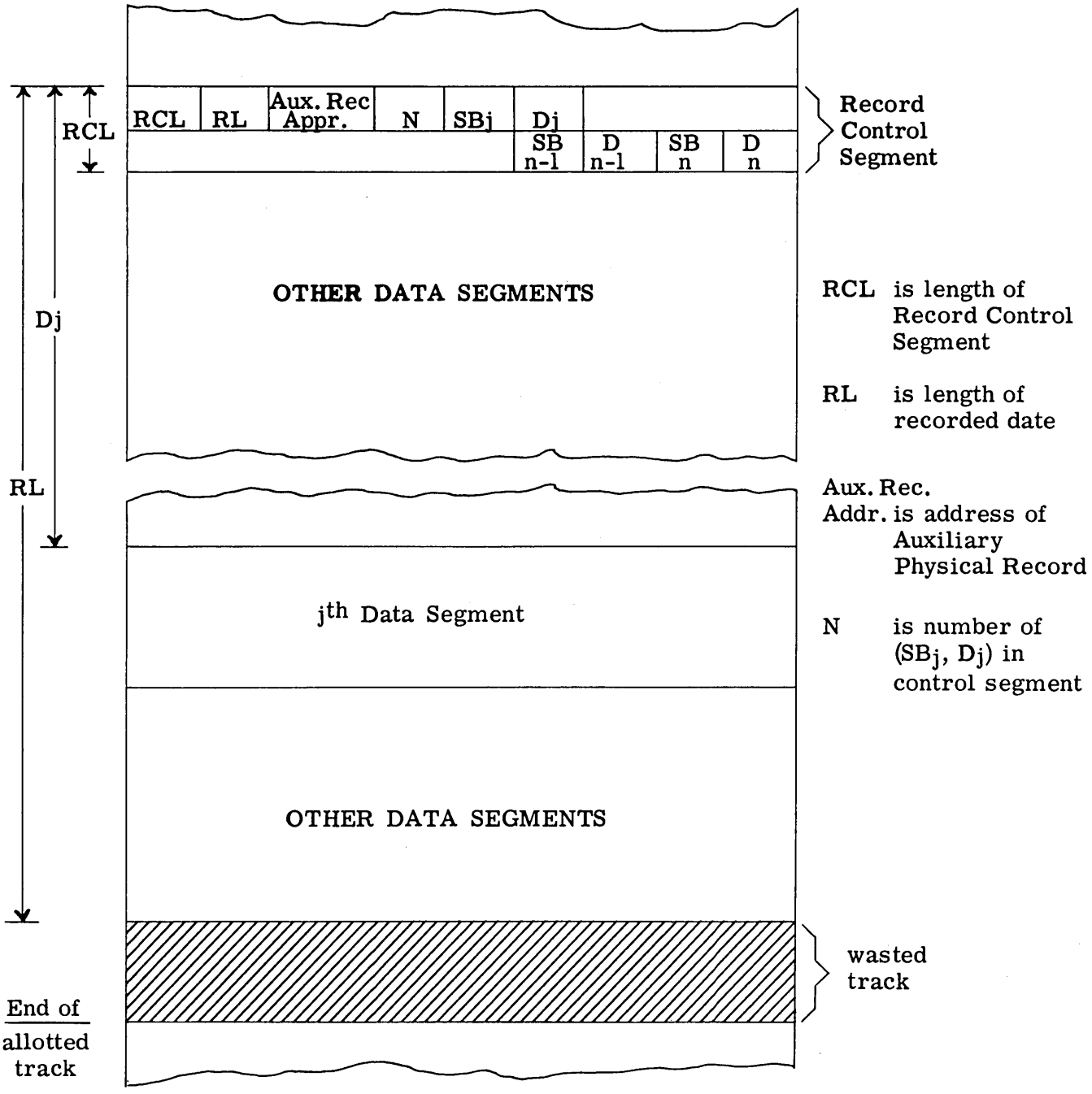


Figure 8—Memory map of physical record

In many business applications there is a constant need for studies of the file records. Frequently the records desired are not known specifically, but will meet certain criteria. This is the need for an associative capability. By placing the Record Control Segment in the Key Area of the record, the records can be located using a simple key scan program. This way the entire record does not need to be brought into memory. If the Record Control Segment is not placed in the Key Area, the entire record must be brought in, but the logic remains the same. By using the Segment Bits (SB_j) Figure 8, in a register, simple Boolean Statements will identify the desired records without lengthy special coding of "compares and branches."

There are many ways in which the contents of the data base in a MIS can be developed. Unless a disciplined approach is used,[15] subtle duplication or omission of data may result. Earlier, it was mentioned that the mortality of special purpose files is expected to be high. This is in part due to the changing needs of the business enterprise. New information must be entered into the data base. There is always a healthy concern for existing programs when the file is changed. By having the file record itself contain information about where data are located and making it relative, the impact of change is kept to a minimum.

Most programming systems will provide for the "linking" together of various subroutines. Since we can have a general way in which data can be extracted from a file record, it is natural to have a generalized routine written once. On the project where these techniques were used, the name used for this routine is File Interface Program (FIP). This program requires only the key to the desired record and a mask of the requested data. It is written in "tight" coding, which translates an efficient but complex file record into a simplified record for use of programs written in COBOL. It relieves the problem programs of all file reading and writing responsibility. Problem programs need no knowledge of specific locations of file data. As the system evolves, the value of FIP is expected to be even more apparent.

Application comments

Information about BIS is generally available[16] and has been discussed at several industry conferences. In the Pennsylvania Company, some of the phases of BIS have already been implemented. The techniques described here were developed to design the file for BIS. Earlier in 1967, the first real time aspects of the system were placed in service on a trial basis. The File Interface Program, written in assembler language, is serving the inquiry programs that retrieve data on customers' accounts for the use of Service Representatives. This inquiry takes place while the representatives are conversing with the customers, so there is a need for a fast response. The file system is list structured. Some data sets are "threaded." Others are based on the Multi-List System or on the Inverted List technique depending on the operational characteristics which are required. The management of the keys and the interior of the record makes this diversity possible. Presently, the file system contains only those segments which are required for the real time and batch processes active today. As additional segments enter the file due to new work to be done, FIP should preserve the investment in existing application programs.

These problem programs were written in COBOL to give a high degree of transferability. The data mode and organization of the file record is one that conserves storage, but is not suitable for direct use with COBOL. Therefore, the interface program edits and code translates the data to an eight bit mode. Initially, the file device used for the massive data base is the IBM 2321 Data Cell. When or if there is an increase in the use of the file beyond the performance of this device, the same records will be transferred to a faster device without serious reorganization. The record length will be chosen in accordance with the principles discussed, but there will not be changes to existing programs. In fact, due to shipping schedules, initial program testing was done on an IBM 2311 Disk using the same record discipline. An additional advantage of the optimizing techniques was in the form of providing input to the simulation models of the file system. The design loop was one of data collection, preliminary file design, simulate, analyze results, modify design, simulate, analyze, etc.

These techniques have served well. The massive files enjoy a Storage Utilization of 65%. The actual record length of the Main Physical Record is down to 796 bytes; from early designs, not using these methods, of 1800 bytes. A more favorable balance between file processes and application program time exists. This ratio is (1: 1.2). The informal objective of a mean response Time of 10 seconds for inquiries has been met. The distribution of response times is shown on Figure 9, Inquiry Response Times. This data is taken from a GPSS (General Purpose Systems Simulator) model that was used in the design effort. It has been validated by empirical measurements of reality and is deemed conformal.

Fundamental thinking still takes place in the whole problem of data management. The functions performed by FIP seem to lead to a solution using either:

1. **Macro Statements**—given in the application programs that would be assembled along with problem coding. An expanded GET. (Example: GET File Name, Record Key, Segment Name 1, Seg-

ment Name 2, Segment Name n.)

2. **Read Only Storage**— the coding of FIP could be placed here with obvious savings. In effect, a privileged command, or ROS MACRO

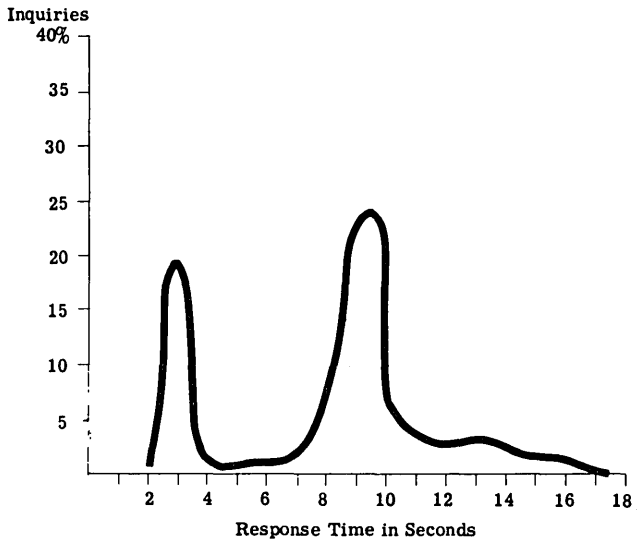


Figure 9—Inquiry response time

It is felt that File Spread Sheet Analysis or intuition would not have produced results as satisfactory as the current design. These techniques are the only ones that seem to properly consider data length, probability, and activity in determining the record design. Other schemes fail to consider them all, or treat some of them casually.

ACKNOWLEDGMENTS

The work described here represents the contribution of many other than the author. Much of the numerical analysis work was built on a solid base provided by Mr. George Pan, of the General Electric Company, Schenectady, N.Y. Messrs. Robert Lanham and Jack Lauback of IBM at Kinston, N.Y., contributed many ideas or participated in application of these techniques. Mr. Richard S. Kaufman of IBM at Philadelphia, Pa. provided the needed counsel for the associated simulation effort which was imbedded in much of this work. Mr. James Weisbecker of the Bell Telephone Company of Pennsylvania was responsible for the demanding application of the files and the development of FIP as a working program. This recognition can only understate the value of their individual talents to the work reported here.

REFERENCES

1 R V HEAD
Real-time business systems
Holt Rinehart & Winston Inc New York p 78, 79 1964

2 W H DESMONDE
Real-time data processing
Prentice-Hall Englewood Cliffs N J p 107 1964

3 C ALEXANDER
Notes on the synthesis of form
Harvard University Press Cambridge Mass p 1-5 1964

4 G J SIMMONS
Application of an associatively addressed distributive memory
AFIPS Proc vol 25 1964

5 E W FRANKS
A data management system for time-shared file processing etc
AFIPS Proc vol 28 1966

6 G S PAN
Linear programming method for optimum file design
Unpublished Master's Thesis Syracuse University N Y 1965

7 F E CROXTON D J COWDEN
Applied general statistics
Prentice-Hall Englewood Cliffs N J p 302-303 1955

8 P M MORSE
Queues, inventories and maintenance
John Wiley & Sons New York N Y p 9 1958

9 W W PETERSON
Addressing for random address storage
IBM Journal of Research and Development vol 1 no 2 1957

10 W P HEISING
Note on random addressing techniques
IBM Systems Journal vol two 1963

11 *The weighted record analysis program users guide*
Un-numbered Communications Industry Marketing IBM Corp White Plains N Y 1966

12 *Introduction to IBM System/360 direct access devices and organization methods*
C 20-1649-0 IBM Corp White Plains N Y 1966

13 *Random access devices series references manual*
70-06-500 RCA Camden N J March 1966

14 *Series 200 summary description—file storage units NP—7099*
Honeywell Inc Wellesly Hills Mass 1965

15 J C MILLER
Conceptual models for determining information requirements
AFIPS Proc vol 25 1964

16 *Business information systems*
American Telephone and Telegraph Company New York N Y 1965

The planning network as a basis for resource allocation, cost planning and project profitability assessment

by H. S. WOODGATE
International Computers and Tabulators Limited
London, England

INTRODUCTION

The planning network is well established as a tool for the planning of projects, and its usefulness as a catalyst for both time and cost control has been demonstrated. However, the task of management involves consideration of other factors besides merely time and cost. The overriding objectives of most commercial organisations are to make a profit, to maximise that profit and to obtain that profit from the deployment of available (or acquirable) resources. The basic planning problem is, therefore, how to decide between alternative ways of using resources such that the best profits are obtained.

These decisions are perhaps the most difficult that management have to face. It is often necessary to make decisions which will affect a significant proportion of the organisation's money flow for many years ahead and to make those decisions at a time when there is a paucity of sound information upon which to base them. Previous experience is often used as a basis of evaluation but as technology advances new projects become more complex and previous experience has less relevance. Simultaneously, competition becomes more fierce and it is increasingly necessary to evaluate cash flows accurately so that adequate (but not uncompetitive) profit margins can be maintained.

It is particularly true of projects where the investment and revenue is spread over several years that the traditional methods of comparing average annual income with total investment are no longer sufficiently accurate to guide the present day decision taker. Project evaluation methods are required which take account of time in conqunction with investment and revenue, so that investment decisions can be made based upon a realistic appraisal of the relative economics of alternative policies.

Two developments have recently appeared upon the commercial and industrial scene in answer to the demand for more accurate financial evaluation of projects. They are the perfection of planning networks, or arrowed diagrams, as a basis for resource allocation and profit planning, and the widespread use of electronic computers capable of rapidly analysing these networks. The logicality of the planning network and the speed of computer processing enable management to test alternative plans and different degrees of planning detail before embarking on major forward planning decisions.

It is, however, unrealistic to expect a computer program to give in a single calculation, a "black box" type of solution to a large complex planning problem involving many estimates and many imponderables. The computer will predict (according to the rules it knows), but management judgement can often enhance these predictions by guiding the calculation along preferred paths. For this reason, the most effective way known so far towards accurate project profit forecasting involves a combination of managerial judgement (to specify the areas of uncertainty) and computing power (to analyse the impact of uncertainty upon factual data). In such a system the computer becomes an extension of the manager's intellect and is used in the way that a manual worker uses a power tool.

This approach marks an important distinction from some earlier experimental systems in this field, where the

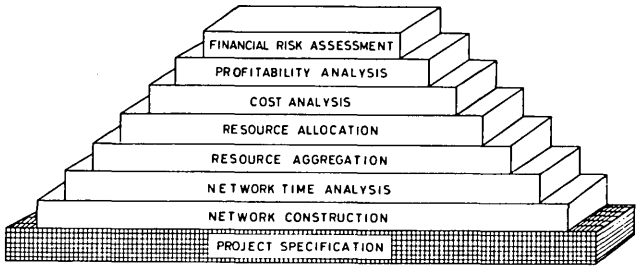


Figure 1—The planning systems pyramid

computer program attempted to provide one single answer, the value of which entirely depended upon the relevance of the programmed decision rules to the project situation being analysed.

In this paper, an approach to profitability assessment is described through the stages of resource allocation, cost planning and profit evaluation. In the system described, the managerial decision process and risk assessment interacts with the preset programmed decision rules in such a way that solutions are reached which exploit to the full any innate managerial wisdom. This method can be considered as a step by step approach and the various stages are as shown in Figure 1. The foundation of this "planning system pyramid" is, of course, the project specification and progress towards profitability and financial risk assessment proceeds through the steps of network construction, time analysis, resource aggregation, resource allocation, and cost analysis. Many of these stages interact and in some cases the inter-action can be delegated to preset decision rules. However, it is a main premise of this paper that many of these inter-actions are managerial decisions and as such should properly be under the direct control of project management. The early steps on the pyramid have been amply described elsewhere and this paper concentrates mainly upon resource, costs and profit planning.

After setting up the planning network and time analysing it in the usual way, the next step is then to consider alternative ways in which resources can be deployed to achieve the physical completion of the project.

Planning resource requirements

The planning network will have yielded at this stage a preliminary estimate of the total project time required and the criticality of individual activities indicates ways in which improvements can be made. However, the execution of the project requires the deployment of men, machines and materials and these invariably fall into a hundred or more non-interchangeable categories. Often too, the amount of resources available is limited, or for economy reasons must be held at a steady level of utilisation.

In planning resource requirements, therefore, the following key questions have to be answered:

1. What are the total resource requirements for a project over its duration?
2. What is the minimum delay to the completion of the project when insufficient resources are available?
3. What is the most efficient utilisation of resources to carry out the project in a fixed time?

Inspection of a network time analysis will quickly indicate that, even with resource information available, finding the answers to the above questions is a formidable undertaking without the aid of a computer.

However, once resource requirements have been added to individual network activities, it is a relatively simple matter for a computer to *aggregate* the resources required over the time span of the project, to show the total requirement of each type. If the computer is also given a statement of the resource availability, comparative statements can be produced which indicate the resource requirement/availability situation, thus giving project management the answer to the first question.

Resource aggregation

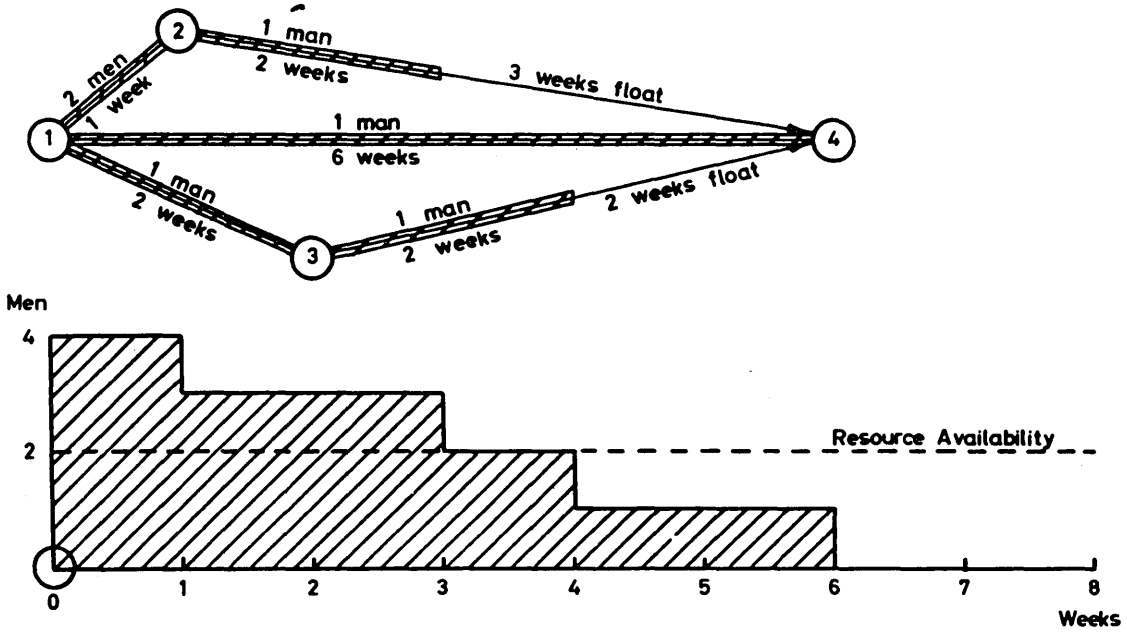
The resources for activities on the critical path are accumulated in the time period allocated to each critical activity. For other activities having float time available, however, there is a choice between aggregating resources at the earliest time or the latest time.

Figure 2 shows the typical effect of these alternatives. In Figure 2(a) the resources required for the simple network shown have been aggregated at the earliest time which can be allocated to each activity and in Figure 2(b) resource requirements have been aggregated at the latest time each activity can be scheduled. A resource availability level is also shown and it is seen that in both cases this has been exceeded.

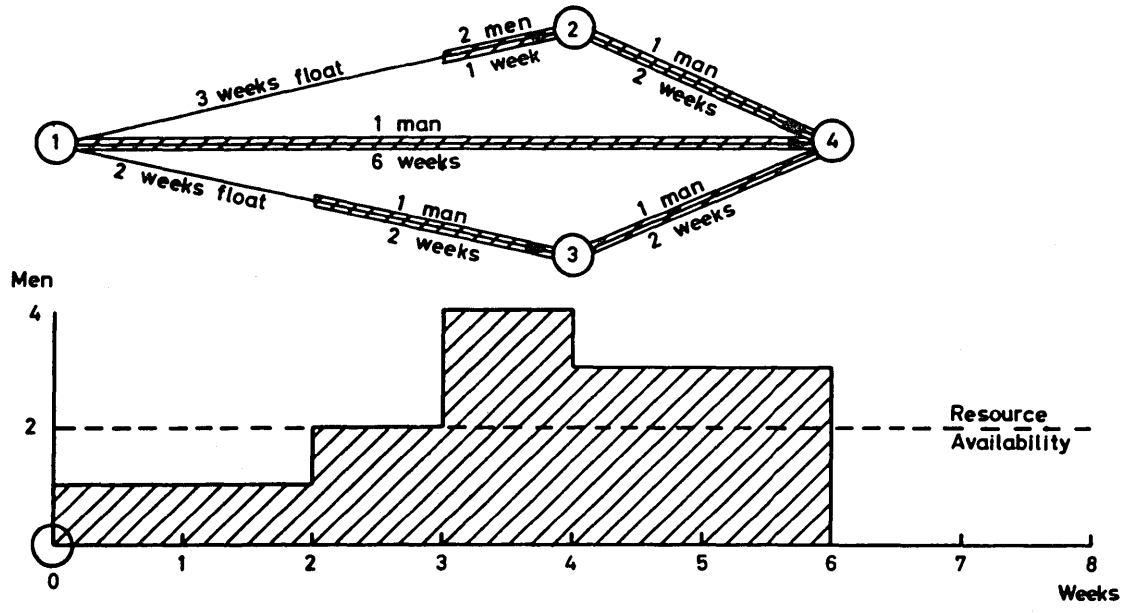
Assuming the extra resource requirements could be met, it would most probably be at some extra cost (i.e., overtime working, sub-contracting or engaging temporary staff). Also it will be noted that available resources are unused for some part of the time and thus further costs are incurred which are additional to the productive labour involved. Both these schedules (earliest or latest) are, therefore, likely to be high cost schedules due to the inefficient use of resources. Thus instead of merely aggregating resources against previously determined time schedules, it is necessary, if minimum costs are sought, to *allocate* the work to be done against the resources available.

Resource allocation

The planning network provides an ideal basis for the allocation of resources, as the network itself defines the sequence in which the work must be carried out and the time analyses show, by the amount of total float, the relative priority attached to each activity. The general method used in resource allocation is to commence at the beginning of the network and prepare a list of all the activities available for scheduling. In the first instance this will naturally be all activities starting from the first event, later in the scheduling process, however, the list of activities available for scheduling will be composed only of activities whose



(a) **Scheduled at Earliest Dates**



(b) **Scheduled at Latest Dates**

Figure 2—Resource aggregation

preceding activities have already been scheduled.

The usual scheduling method is then to allocate the available resources to activities on a day by day (or time unit by time unit) basis. There will be several activities in the list, and as they can only be dealt with one at a time, the next step is to arrange them in order of priority. The squence in which activities are scheduled is important as it is a question of 'first come first served', the first activities having the pick of available resources and the later ones possibly having to be delayed.

It might at first sight appear that critical activities, or activities with small amount of float, should be scheduled first. In practice however, it has been found that there is advantage in scheduling the shortest activities first and other factors also influence the choice of scheduling priority.

As resource allocation proceeds on a day by day basis from the beginning of the network to the end, the situation inevitably arises where there are insufficient resources available in a particular time period and then an activity, or part of an activity, must be carried over until the next time period which has sufficient resources available. In a resource limited situation such as this, the carrying over process will usually extend the project duration and cause the completion date to be delayed.

An example of this is shown in Figure 3(a). Here the network shown in Figure 2 has been allocated on a resource limited basis and in consequence the completion date has to be extended by 1 unit (from 6 to 7). Thus the second question "What is the minimum delay to the completion of the project when insufficient resources are available?" is answered in this particular case.

However, the third question posed above "What is the most efficient utilisation of resources to carry out the project in a fixed time?" is still to be answered. To ascertain this the allocation process is carried out with a fixed time limit and no delaying of activities is permitted beyond the amount allowed by the available float time. In doing so, it is inevitable that the level of available resources is exceeded but it is arranged that this will take place at the point where it makes the minimum increase above the preset level. The effect of the operation is shown in Figure 3(b), where the level of resource availability has been exceeded in two places but the increase nowhere exceeds one unit. Thus, the four schedules, Figure 2(a) and (b) and Figure 3(a) and (b), indicate the main alternatives available for the specimen network.

However, real projects are much more complex than this example. The network may comprise several thousand activities and involve more than a hundred different types of resource. Each network activity may require several different resources (e.g., bricklayers, labourers and cement mixers) to be considered simultaneously. Sometimes a resource may only be required for part of the activity duration, as for example in building a wall where labourers may dig and lay the foundation but bricklayers only arrive after this work has been done.

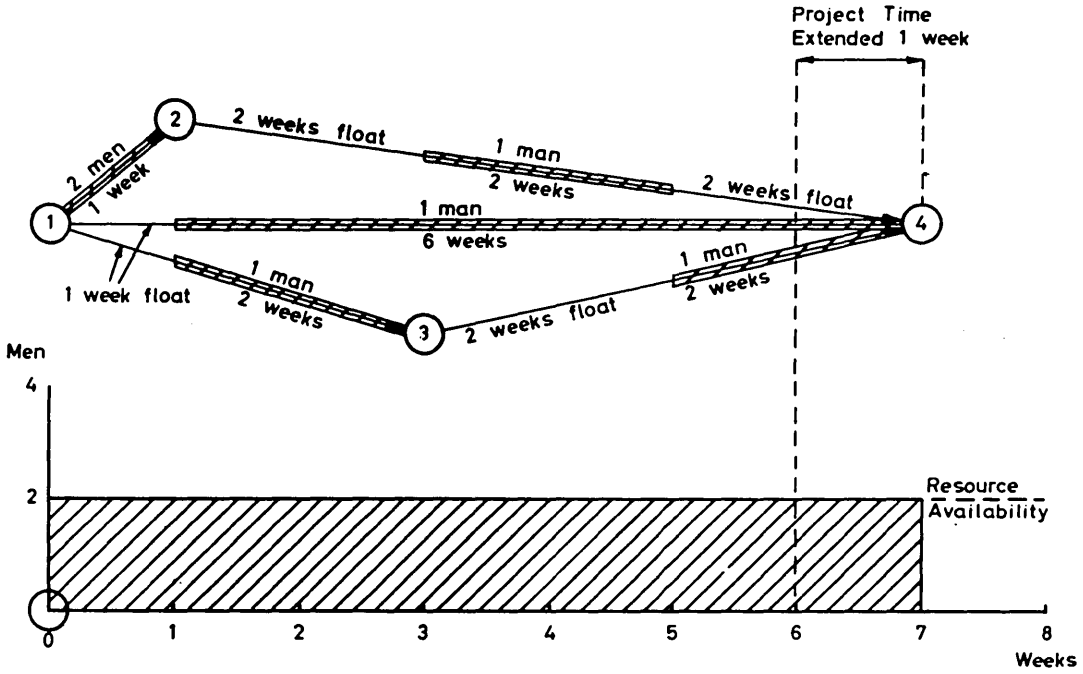
There are often special conditions to be observed when formulating work schedules. For example, some activities once started must be worked continuously without interruption (as laying concrete), others may be interrupted at any time (as building a fence), others can be interrupted only at special times (as between finishing the foundation and commencing bricklaying). Also, resources fall into different categories and require different treatment, for example manpower and equipment are wasting assets in as much that if they are not used then productive capacity is lost forever, Materials on the other hand, if not used, can be stored and used the next day without waste. Some materials (like concrete) are consumed and others (like scaffold tubes) can be reinstated as available resources after use.

Availabilities of resources too seldom follow the constant line shown in Figure 2. They are usually cyclical with five or six days of work and two or one day (weekends) of no work. Some industries have lower resource availabilities on Mondays and Fridays due to absenteeism. Thus each week comprises not one but four levels. Public and annual holidays add further complexity to the scheduling process.

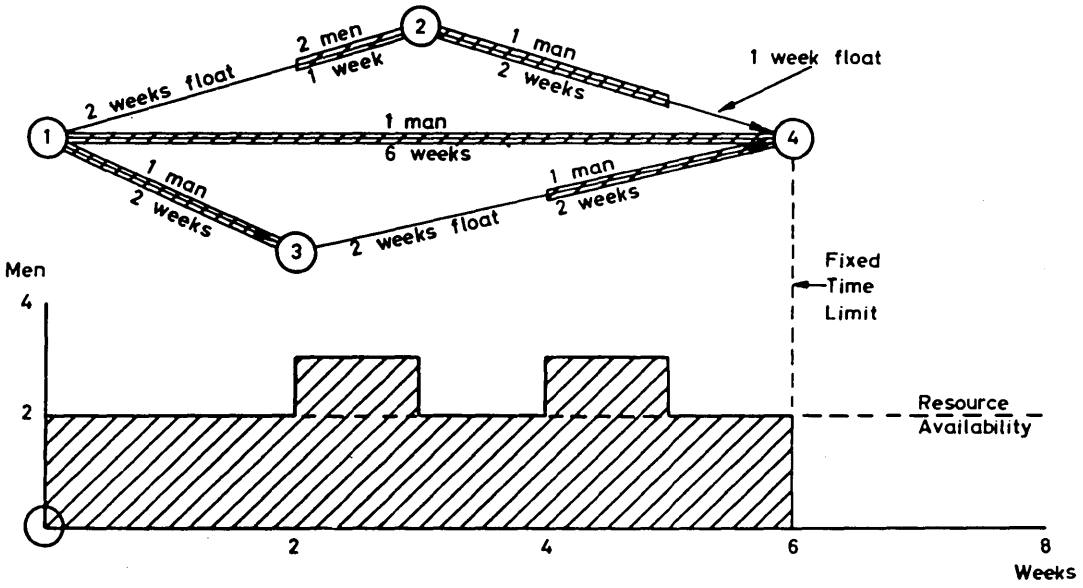
Additionally, resource levels themselves are not finite and absolute. The initial prognosis may be to only work normal hours, but overtime working is an established part of the current industrial scene. Thus any practical approach to resource allocation must take account of the possibility of employing labour and equipment for a greater time than the normal working hours—overtime will probably be worked anyway!

When resource scheduling, therefore, it is usual to specify two levels, firstly, the normal level and secondly the maximum level. This second level is usually the maximum possible overtime and in the terminology of resource allocation is known as the "resource threshold." The cost of resources within the threshold will usually be higher, a point having significance in project cost analysis, which is described later.

To complete this survey of the factors to be considered in resource scheduling, it is necessary to consider the significance of the project end date. Like resource levels, the end date is not usually fixed and unalterable, some flexibility is usually present in as much that project completion is acceptable over a range



(a) Allocation within resource limit



(b) Allocation within time limit

Figure 3—Resource allocation

of dates. This is especially true at the initial planning stage where contractual commitments have not yet been established. When it is realised that the plan eventually adopted will inevitably be a compromise between early completion dates, efficient resource utilisation and economic costs, it becomes apparent that flexibility of completion date is the only means of obtaining suitable arrangements in terms of resources and costs.

A method of expressing flexible end dates is to set a "preferred completion date" and a "maximum permissible completion date". In the jargon of the trade, the difference between these two dates is known as the "project duration threshold."

In this brief survey, the principle factors to be considered have been described sufficiently to highlight the somewhat overwhelming complexity of the resource allocation task. Fortunately considerable headway has been made in recent years in development of computer programs to undertake this calculation. It must be said, however, that the mathematics for a *perfect* solution does not exist and it is still necessary for management to exercise judgement in the specification of scheduling objectives, priorities, the validity of basic data (networks and time/resource estimates) and approval of computed results. Here the senior manager is fulfilling his traditional role of giving instructions to his planners and approving their work without necessarily being able to check the calculations in detail. Using a computer, however, he is assured that his instructions (inherent in the computer program) are obeyed exactly and clerical mistakes will be virtually non-existent.

A computer can very quickly calculate the best manner of allocating available resources so that particular project objectives can be met within the priorities set by management. In these complex situations, a computer will ascertain the best solution, but it often happens that the basic problem described so far is insoluble when the resource requirement exceeds the resource availability and the project duration is considered to be fixed. The adage "you cannot put a quart into a pint pot" is particularly relevant to this situation and it is necessary to exercise a deliberate choice of the manner in which the resource requirement shall overflow the resource availability. The choice is usually between delaying the completion date, using more men and machines, or working extra hours. Any of these choices will usually have maximum limits (thresholds) which it is not possible to exceed.

The alternatives are shown diagrammatically in Figure 4. Here the resources (vertical scale) are plotted against project duration (horizontal scale). Referring to Figure 4, it is seen that the resources have a normal level (i.e., the most economic rate of working) and above this the resource "threshold" representing an additional capacity which can be obtained at additional cost (i.e., overtime working, double shift working, extra plant, etc.). The threshold has, however, a finite capacity which cannot be exceeded. Also shown beyond the preferred completion date on the horizontal scale is a project duration threshold up to the maximum permissible completion date. The project duration threshold represents the tolerable delay (if any) which can be permitted under dverse resource availability situations.

Figure 4(a) shows the simple case where there are adequate resources for the completion of the project by the preferred completion date and Figures 4(b), (c) and (d) illustrate the alternatives when resource availabilities are insufficient. In Figure 4(b) the resource threshold has not been used by the project and the completion date is extended. In Figure 4(c) the completion date has been maintained but the (higher cost) resource threshold has been utilised. In Figure 4(d), both the resource threshold and the project duration have been taken up.

A further case exists, of course, where even if the resource threshold is used the project cannot be completed by the maximum permissible completion date. Usually in this event drastic action is necessary in respecifying the project.

Consideration of the different scheduling alternatives involves an iterative analysis of the project with differing restrictions. The computer is particularly useful at this stage, as it can produce a variety of simulations of possible alternatives and print the results in a form which can be quickly assimilated by those responsible for taking decisions about the project plan.

Examples of computer analyses are shown in Figure 6 which is a resource histogram of one trade on the network Figure 5. In Figure 6(a), the computer has constructed a schedule which contains the entire work content within the normal resource level. This has delayed the completion date from 4th January 1967, to 23rd March 1967. In Figure 6(b), however the computer was instructed to schedule within the fixed time (4th January 1967) given by the original time analysis. Here it is seen that the work can be achieved but this particular resource is utilised right up to the threshold level for two weeks and within the resource threshold for five weeks.

The computer has produced clear unequivocal statements of alternative policies but it is still the prerogative of project management to decide between them. Further guidance will obviously be helpful and fortunately this can be extracted from the planning network. As the differing scheduling alternatives give different project durations and project costs, it is useful to consider the alternatives in conjunction with project cost analyses before selecting the plan to be implemented.

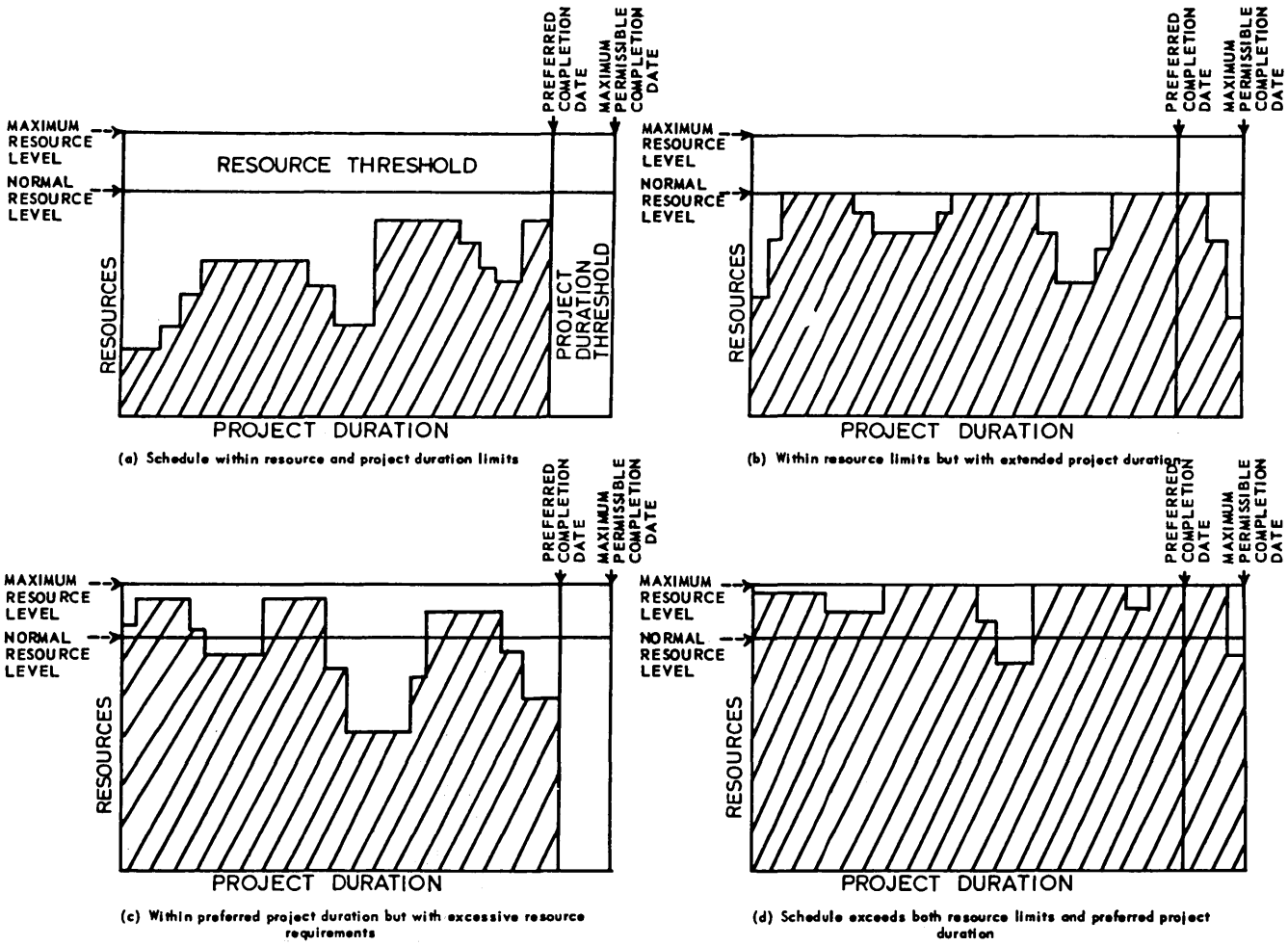


Figure 4—Resource and project duration thresholds

Project cost analysis

The next step of progress towards an acceptable project plan is to carry out a cost analysis of the network. When performing this, it is convenient to consider project costs in two elements viz.:

Direct Costs
Indirect Costs

Direct Costs are those expenses which can be attributed directly to individual activities and are usually obtained by applying the appropriate rate to the resources necessary for the activity. Hence labourers may be priced at £3 per day and bricklayers at £5 per day for normal working. Where premium (or threshold) resources are available, then the appropriate higher rate will also be specified. During resource allocation, the aim is to minimise the amount of premium resources used and hence the lowest cost schedule will be produced automatically. For cost analysis purposes, the amount of each resource used is extended by the rate which is applicable, i.e., normal or premium and the amount accumulated for each time period covered by the network.

Indirect Costs arise from the practical difficulty of apportioning some costs accurately to individual activities. Such things as administration, storekeepers, security and other overheads cannot properly be defined as relating to particular activities but rather to groups of activities or even the entire network. The network planning method of handling these costs is to introduce special activities which are for the sole purpose of spreading indirect costs. These will start and finish at the events on the network where the particular indirect costs are deemed to start and finish and the money involved is applied to these activities either as a rate per day which is then built up over the calculated duration between the two events, or as a lump sum which is spread over the duration of the activity. No time is specified for these "cost only" activities, as their duration is calculated during the analyses of the network.

A further use of "cost only" activities is to provide a means of bringing project delay costs into the calcula-

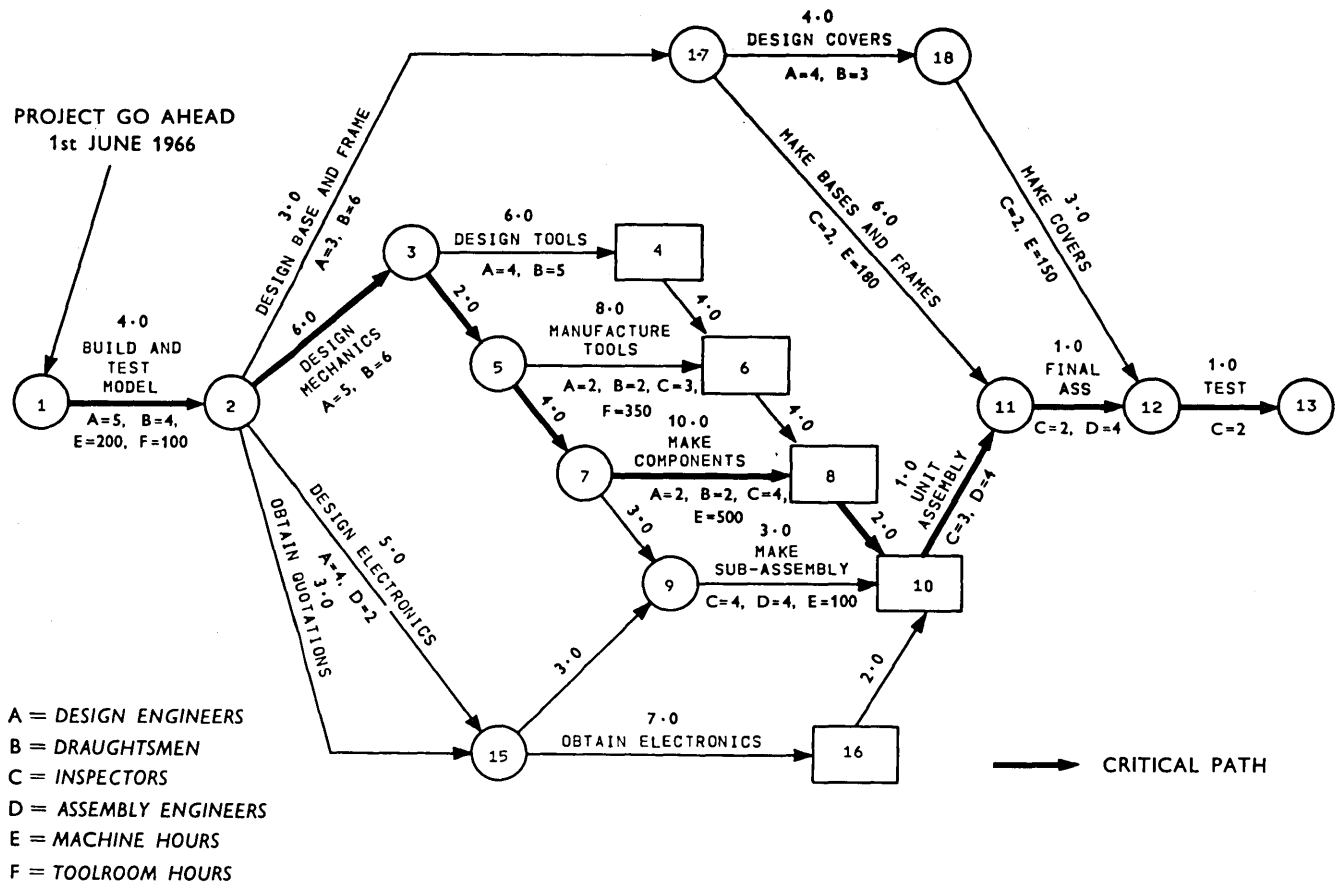


Figure 5—A small planning network

tion. Contract penalty clauses are an obvious application of this technique, but it will apply equally to the cost of lost revenue occurring through non-completion on certain dates, i.e., loss of rents of buildings, loss of production when plant is shut down for maintenance, loss of income from delayed opening of shops, etc.

To complete the build-up of project costs, the direct and indirect costs are accumulated to obtain the total expenditure over the duration of the project. It is customary to show the project cost analysis as a cumulative curve as in Figure 7, from which it is possible to see at a glance the total expenditure required and also the way in which the rate of expenditure is spread over project duration. The calculations involved in this analysis are not particularly complicated, providing the basic data is available, and if a computer is used for the analysis of the network and the allocation of resources, then cost curves (as shown in Figure 7) can be printed direct from the line printer of the computer.

It will be apparent that, with the range of factors considered in this cost analysis, both spread of cost and total cost are influenced by the work schedule produced during the resource allocation process. The direct costs will vary according to how much higher-cost, or threshold, resources have been used and some indirect costs will rise with increase in project duration.

Examples of the variation which may occur are shown in Figure 8. Here are shown alternative cumulative cost curves which could arise when differing resource allocation policies are applied. In Curve No. 1, the project was scheduled within a fixed time limit which caused high-cost threshold resources to be used and hence produced a high-cost schedule, requiring £19,-600. In the case of curve No. 2, a schedule was calculated within normal resource availabilities and thus threshold costs were avoided. The project completion, however, is delayed and as the project bears some indirect costs which are proportional to time, these continue to rise during the extra project time and more than offset the saving accruing from the avoidance of resource threshold costs. The combined effect is to produce a total project cost of £21,200 which is higher than alternative No. 1.

Dependent upon the rate and time displacement of indirect costs, however, the cost analysis could well have resulted in curve No. 3. Here the indirect costs do not offset the savings in direct costs (resulting from the avoidance of threshold resource costs) and the

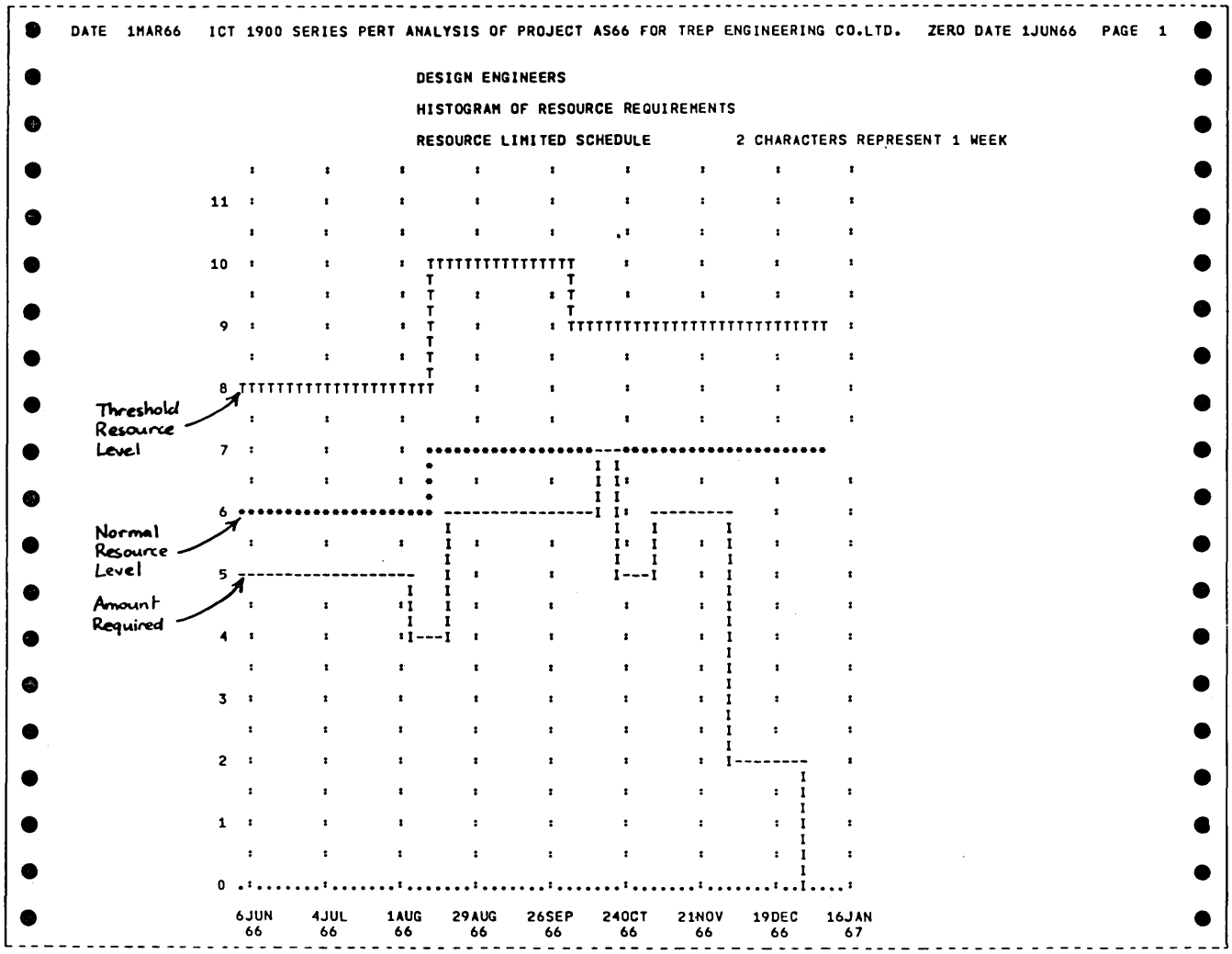


Figure 6(a)—Resource histogram—resource limited

combined effect is to produce a lower overall cost (£17,800) for the completed project.

Using techniques of network based resource allocation and cost analysis, project management have available a convenient method of realistically evaluating planning alternatives. It will, however, be apparent that a considerable amount of calculation is involved to produce a number of alternative schedules and cost analyses upon which management can adjudicate and the computer is particularly useful as a means of alleviating this chore. The fact that the modern computer can produce such information in graphical form considerably aids communication between management and planning personnel. An example of a computer produced diagram is shown in Figure 7.

Project profitability

Having assessed in some detail the amount and spread of costs which the project is likely to incur, it is next necessary to make similar estimates of the income and any other cash movements which may be associated with the project before profitability can be analysed. To assess profit accurately, it is necessary that the whole of the life of the project be considered and all associated incomes and expenditures be included. Thus complete analysis will involve continuing operating costs, revenues, investment grants, taxation and tax allowances.

An important factor associated with these estimates is their displacement in time relative to the initial project expenditures. The planning network can be used to obtain these displacements merely by inserting the appropriate costs (suitably identified) into the network at the appropriate point and by extending the network (if necessary) with "cost only" activities beyond its normal completion.

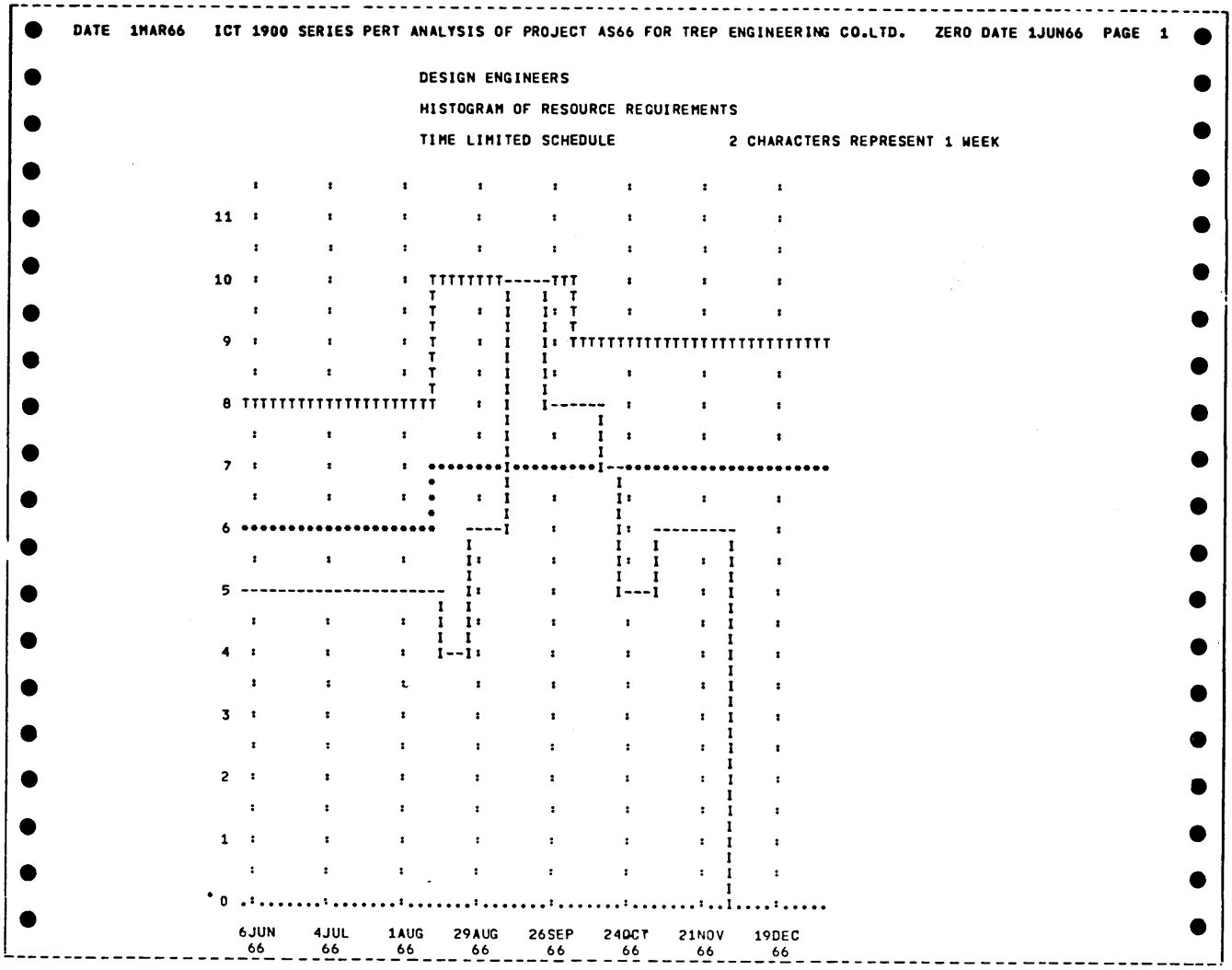


Figure 6(b)—Resource histogram—time limited

A further factor in this analysis is an assessment of risk, as future profits can never be free from uncertainty. The degree of risk, however, will vary and thus the quantitative assessment of the degree of risk is a necessary adjunct to the consideration of the possible amounts of profits. Risk arises because of uncertainty inherent in the various estimates made during the analysis. Thus an objective financial assessment of a project must take account of the following main factors:

1. Amount *and timing* of investment
2. Amount *and timing* of income
3. Possible variations in both investment and income.

Cash flows

The amount and timing of both investment and income can conveniently be considered as cash flows, whereby an expenditure represents an outward flow (negative) and an income represents an inward flow (positive). Uncertainty about either inward or outward flows can then be represented by alternative values for these items.

The use of network planning techniques as a basis from which to build up realistic work schedules and accurate assessments of project costs have been described earlier. The cumulative cost curve of the project plan selected shows the proposed expenditure over time and thus provides the basis for one of the principle cash flows involved in profitability assessment. For a complete evaluation, however, it is necessary to consider all other expenditure and incomes which will be relevant to the project.

An example of such a study (made by computer) is illustrated in Figure 9. Here the building of a factory extension is under consideration. The plan envisages that this will be built over a period of 2 years 3 months (between 1966 period 3 and 1968 period 3) special

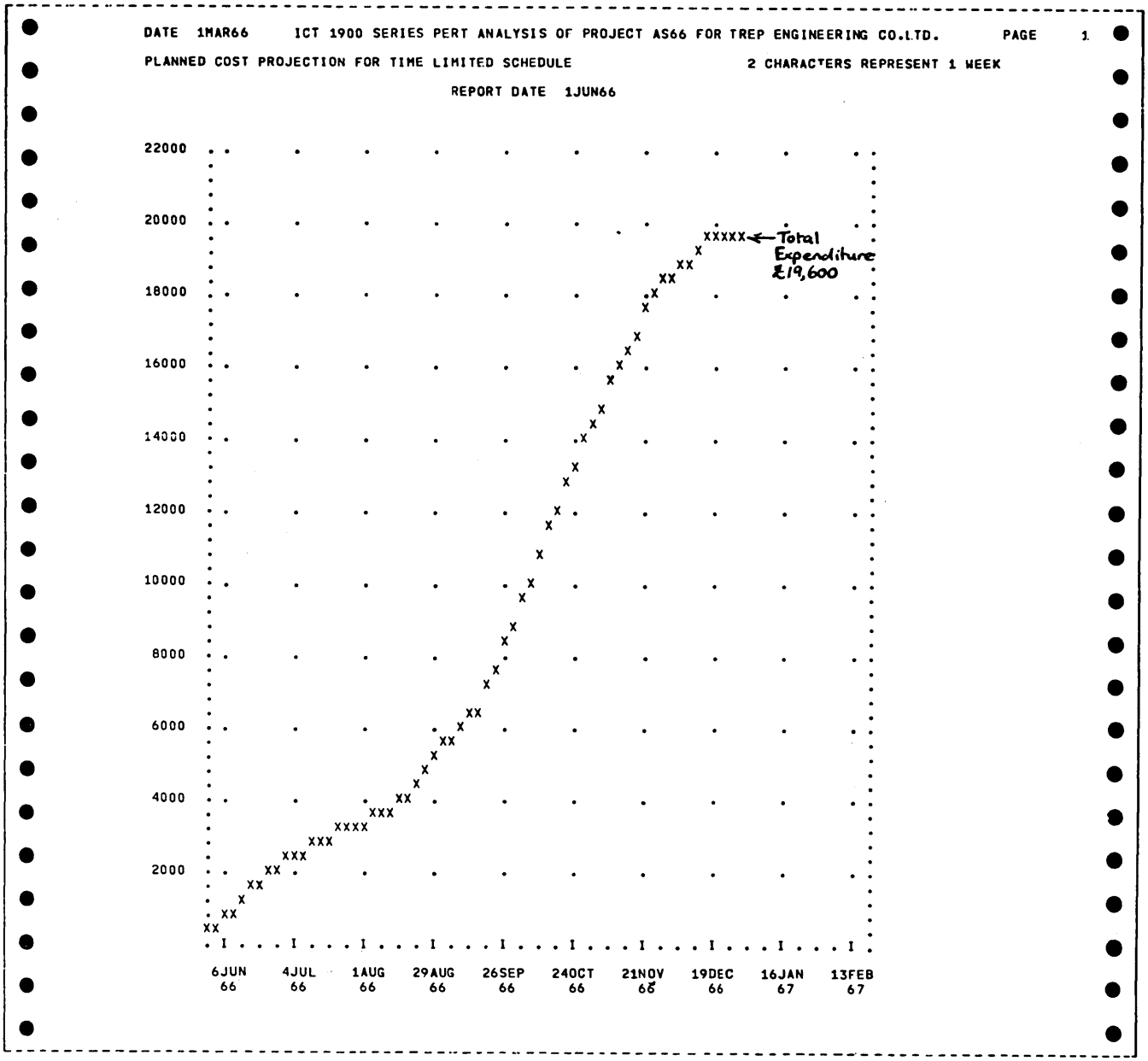


Figure 7—Project cost curve

equipment to be installed will have a life of 4 years (between 1968 period 2 and 1972 period 1). The total cost of the factory and plant will be £270,618 and this will be depreciated over 4 years, after which it will have a scrap value of £2,000 (in 1972 period 2). There is also another small recovery of £500 (in 1971 period 1) due to the interaction of another project which will make part of the plant redundant. The initial cost and the two recoveries are shown in Figure 9 in the columns headed "CAPITAL INVESTMENT" and "OTHER CAPITAL." As described earlier, the negative sign indicates expenditure and no sign indicates income.

Investment grants, initial and annual tax allowances are significant items in the financial evaluation of a development project, the incomes accruing from these have been calculated by the computer and are shown separately in Figure 9 in the columns marked "INVESTMENT GRANT" "INITIAL ALLOWANCE" and "ANNUAL ALLOWANCE." These items are subject to delayed payment—this delay has been calculated and the amount entered at the date at which the cash value of the allowance will be received. The negative annual allowance figure is a refund caused by the recovery of the scrap value of the project.

In order to ascertain the income which will be ob-

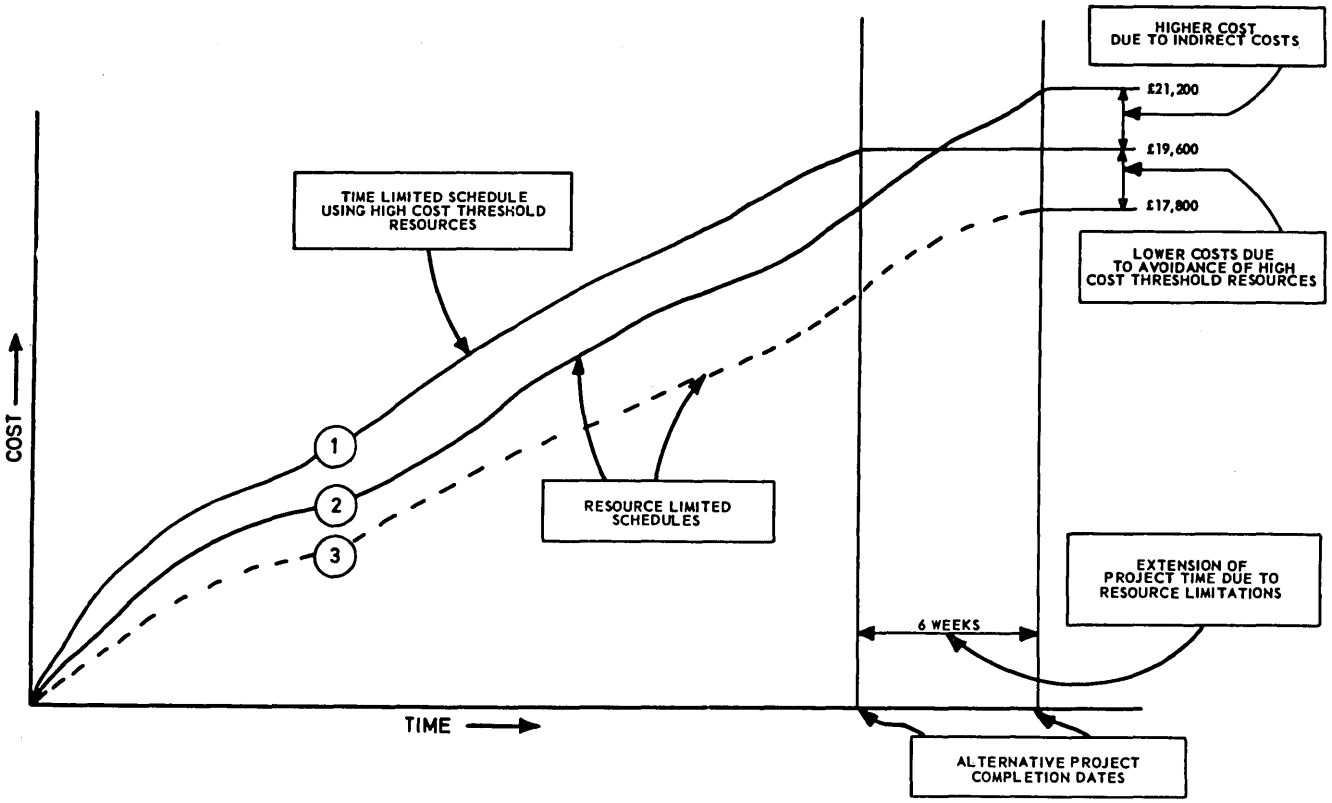


Figure 8—Alternative cost plans

tained from the project, estimates of the operating costs and revenue are included in the appropriate time periods. The income derived from the project is shown under the heading "REVENUE INCOME," and the routine cost of operating the plant and using materials, labour, etc., is shown unded "REVENUE COST." The computer makes the simple subtraction to give the incomes shown under "REVENUE PROFIT." Here it will be noted that the first operating period (1968 period 2) is expected to make a loss (-5000) during the time the plant is working but not yet producing goods.

"The tax man cometh!" In the next column on Figure 9 marked "TAX ON PROFIT" the amount of corporation tax payable on the revenue profit shown in the previous column is calculated. The cash value shown in this column is entered after the appropriate time lag at the actual date the tax payment is made. With this information, the "NET CASH FLOW" is calculated. This is the difference between the cash inflow (income) and cash outflow (expenditure) and is the arithmetic sum of the capital investment and the various allowances, the revenue profit and tax on profit. This column indicates the amount of money which is to be put in or taken out during each time period of the project.

In the next column of Figure 9 an increment has been included for "INTEREST" on the cash required to finance the project. In this example, it has been specified that cash required will attract an added interest rate of 7% per annum (amount indicated by a minus sign) and when the project accrues surplus funds, then this can be re-invested at 4% per annum. These percentages are selected as representing the borrowing and lending rates open to the company at the time. The net cash flow and interest elements are now combined to give the "CUMULATIVE CASH FLOW," shown in the last column. This total shows the position of the investment at the end of each successive period.

Financial evaluation of project

The cumulative net cash flow for the project shows the pattern of increasing expenditure during the initial period of the project and how recovery is effected. It shows the point at which the cumulative income exceeds the cumulative expenditure (the project duration to this point is usually known as the *payback period*)

FINANCIAL EVALUATION OF PROJECT

EXTENSION BLACKMORE PLANT, DEVLPMENT AREA - 25% GRANT, 15% INITIAL, 20% ANNUAL

STR. LINE. NO COST/INCOME IN/DEFLATION. INCOME EST. ON 5% ANNL. MARKET GROWTH.

YEAR	PERIOD	CAPITAL INVESTMENT	OTHER CAPITAL	INVESTMENT GRANT	INITIAL ALLOWANCE	ANNUAL ALLOWANCE	REVENUE INCOME	REVENUE COSTS	REVENUE PROFIT	TAX ON PROFIT	NET CASH FLOW	INTE REST	CUMU-LATIVE CASH FLOW
1966	3	-3766.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-3766.0		-3766
	4	-7832.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-7832.0	-64	-11662
1967	1	-51800.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-51800.0	-198	-63660
	2	-12317.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-12317.0	-1082	-77059
	3	-33718.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-33718.0	-1310	-112087
	4	-60214.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-60214.0	-1905	-174206
1968	1	-83261.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-83261.0	-2961	-260429
	2	-15567.0	0.0	0.0	0.0	0.0	20000.0	-25000.0	-5000.0	0.0	-20567.0	-4427	-285423
	3	-2143.0	0.0	0.0	2709.6	0.0	35000.0	-32000.0	3000.0	0.0	3566.6	-4852	-286709
	4	0.0	0.0	0.0	0.0	0.0	50000.0	-35000.0	15000.0	0.0	15000.0	-4874	-276583
1969	1	0.0	0.0	0.0	0.0	0.0	60000.0	-30000.0	30000.0	0.0	30000.0	-4702	-251285
	2	0.0	0.0	15053.5	0.0	0.0	70000.0	-27000.0	43000.0	0.0	58053.5	-4272	-197504
	3	0.0	0.0	18565.2	4042.3	9002.5	70000.0	-27000.0	43000.0	-5200.0	69410.0	-3357	-131451
	4	0.0	0.0	3891.7	0.0	0.0	70000.0	-27000.0	43000.0	0.0	46891.7	-2235	-86794
1970	1	0.0	0.0	0.0	0.0	0.0	70000.0	-30000.0	40000.0	0.0	40000.0	-1475	-48269
	2	0.0	0.0	0.0	0.0	0.0	70000.0	-30000.0	40000.0	0.0	40000.0	-820	-9090
	3	0.0	0.0	0.0	0.0	9002.5	70000.0	-33000.0	37000.0	-63600.0	-17597.5	-155	-26842
	4	0.0	0.0	0.0	0.0	0.0	65000.0	-35000.0	30000.0	0.0	30000.0	-456	2702
1971	1	0.0	0.0	0.0	0.0	0.0	65000.0	-35000.0	30000.0	0.0	30000.0	26	32728
	2	0.0	500.0	0.0	0.0	0.0	65000.0	-40000.0	25000.0	0.0	25500.0	311	58539
	3	0.0	0.0	0.0	0.0	9002.5	60000.0	-40000.0	20000.0	-58800.0	-29797.5	556	29298
	4	0.0	0.0	0.0	0.0	0.0	60000.0	-45000.0	15000.0	0.0	15000.0	278	44576
1972	1	0.0	0.0	0.0	0.0	0.0	60000.0	-50000.0	10000.0	0.0	10000.0	423	54999
	2	0.0	2000.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2000.0	522	57521
	3	0.0	0.0	0.0	0.0	11253.1	0.0	0.0	0.0	-36000.0	-24746.8	546	33320
	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	316	33644
1973	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	320	33964
	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	323	34287
	3	0.0	0.0	0.0	0.0	-800.0	0.0	0.0	0.0	-4000.0	-4800.0	326	29812
TOTALS		-270618.0	2500.0	37510.4	6751.9	37460.6	960000.0	-541000.0	419000.0	-167600.0			

TOTAL INVESTMENT (LESS ALLOWANCES) - 186395

TOTAL REVENUE INCOME - 960000

NET PROFIT (AFTER TAX & DEPREC.) - 29812

D.C.F. YIELD RATE OF RETURN - 10.96 % PAYING RATE (4.00 % EARNING RATE)

D.C.F. NET PRESENT VALUE - 22636 AT 7.00 % PAYING RATE & 4.00 % EARNING RATE

AVERAGE ANNUAL RATE OF RETURN - 3.43 %

PAYBACK PERIOD - 4 YEARS 2 PERIODS

Figure 9—Financial evaluation of project

and also gives a general indication of the relative size of expected profits to expected expenditure.

The same information is calculated by the computer and printed on the financial analysis Figure 9, where it is seen that the net profit after meeting all tax commitments and fully depreciating all plant and equipment, is expected to be £29,812. The net profit shown here is, of course, only a guide to the general order of profit expected at the end of the project. It will be seen that the amount is heavily conditioned by the interest rates (7% and 4%) applied to the cash flows. If these rates are varied, then the net profit will also alter significantly. This net profit shown is derived from a net capital investment of £186,395 and a revenue income of £960,000. The maximum cash requirement at any one time will be £286,709 and will occur in 1968 period 3. Also calculated and shown in Figure 9 is the payback period (described above) which is here 4 years 2 periods after the commencement of the project.

Discounted cash flow yield

However, this information does not complete the analysis. The next step is to calculate the financial *yield* (i.e., return on investment expressed as a percentage) obtained from the project. Here it is called *Discounted Cash Flow* (D.C.F.) Yield and follows the 'discounted cash flow' method of calculating financial yield. Discounted cash flow is gaining increasing usage in recent years and is based upon the fact that the purpose of an investment is to obtain a series of future annual incomes over the life of the project. The technique of D.C.F. Yield is to find the rate of interest which discounts future income from the project down to a 'present value' equal to the initial investment. This rate is called the "yield" and is often used as a profitability index for the purpose of inter-project comparisons.

Another way of defining "yield" is that rate of interest which the funds estimated to be required for the project would need to earn if, instead of being used

on the project, they were invested elsewhere at the same time as those estimated for the project, so as to give exactly the same returns at exactly the same time as the project predicts.

This concept of time and value can best be illustrated relative to a table of compound interest i.e., £100 invested at 5% compound interest increases each year as follows:

1st year £100 x 1.05 = £ 105
2nd year = £(100 x 1.05)1.05
= £100 x 1.05^2 = £110
3rd year = £100 x 1.05^3 = £116
4th year = £100 x 1.05^4 = £122

Thus it can be said that the 'present value' of a profit of £122 in four years time is only £100. Similarly the 'present value' of any future profit can be obtained by dividing by a 'present value factor' which is the reciprocal of the compound interest factor e.g. again assuming 5% compound interest, £100 in four years time has a 'present value.' $\frac{1}{105^4}$

= £100 x = £100 x
= £100 x .8227 i.e. present value factor
= £82

There are, therefore, three elements involved—present value, future value, and the compound interest rate which relates the two values. If the two values are considered as cash flows (as in example Figure 9) then the present value is negative (i.e., expenditure) and the future value is positive (i.e., income). The interest rate is then the rate at which discounted positive cash flows equals discounted negative cash flows.

The present value of negative £82 however, shown above, is not discounted (as it is already the present day) and the positive £100 is only discounted for one year. In the larger example Figure 9, both the negative and the positive cash flows occur over several years the present value is negative (i.e., expenditure) and the D.C.F. Yield) applicable to the combined cash flows of several years. The method of obtaining this involves repeated calculation of the present value using different yield rates until a rate is found for which the and it is required to calculate the interest rate (i.e., the sum of the discounted negative cash flows.

Dual rate D.C.F.

In the single rate D.C.F. method just described, the discount rate is considered to be the same when the calculation is on either positive cash flows or negative cash flows. This is not entirely realistic, as it is normal for interest rates to be different according to whether the required cash is being "lent" or "borrowed." To accommodate this distinction, the calculation can be made with different rates according to whether the project is absorbing capital or showing a profit. The two rates are termed *earning rate* and *paying rate.*

Earning rate is the rate of interest which can be obtained on the surplus funds accruing to the project. In the example Figure 9 this rate has been set at 4% but it could be set higher if the money is to be invested more profitably or it could be zero in the unlikely event of the cash not being used.

Paying rate is the rate of interest which the project must pay in order to obtain the funds necessary to finance the project. This could be the market rate (e.g., 7%) or if it is desired to calculate the yield, it could be the D.C.F. rate, i.e., the calculated rate which discounts the present value of the project to zero.

The application of different rates according to the state (i.e., positive or negative) of the investment is usually known as a *dual rate analysis* and where one of the rates (usually, but not necessarily, the paying rate) is the D.C.F. rate, it is termed *dual rate D.C.F. analysis.*

A dual rate D.C.F. analysis has been performed on the cash flows shown in Figure 9 with the earning rate set at 4% per annum and has revealed a D.C.F. yield paying rate of return of 10.96%. This indicates that if the project had not been undertaken, it would have been necessary to obtain that rate of investment in order to earn the same profit from the same amount of money invested over the same time span.

Profitability Analysis

With the information shown on the "Financial Evaluation of Project," management have a comprehensive set of data upon which an assessment of the financial merits of the project can be deduced. Perhaps, more important, because the information has been arranged and analysed in a logical manner, it offers a standard method of making comparisons between the respective merits of different projects or different approaches to the same project.

This consistency of evaluation is invaluable in the process of communication, when several managers are required to discuss the merits of different projects. The facts presented in the analysis are unambiguous and based upon sound mathematical principles but, even if details of the method are challenged, the fact that it has been consistently applied to all projects under consideration still enables the comparative aspect to apply.

Initial evaluation is, however, usually only the starting point for a further study of alternative plans. The project may be speeded up by refining the planning network, more resources may be applied, production costs may be re-examined and alterative marketing strategies (i.e., pricing structure and sales forecasts) may be tested to ascertain the effect upon income. With each set of data, the calculated D.C.F. Yield and associated information give a sound guide to the financial merits of each particular plan.

Financial risk assessment

It will have been noted that the financial evaluation of the project in Figure 9 was based upon a number of estimates about probable expenditures and incomes in the future. Whilst the detailed cost planning will have introduced a good measure of accuracy into part of the expected cash flows, there remains much data which could be subject to wide fluctuation. Actual sales will ultimately depend upon the state of the market at the time that production is possible and this is conditioned by the general economic situation and the actions of any competitors. Similar factors such as the variability of cost of materials and labour could also influence production costs. Any variation in the information used will of course, affect the yield and hence the profitability of the project. Few investment decisions will, therefore, be settled by a single evaluation calculation and the decision taker will certainly want to investigate some of the alternatives which might arise.

Whilst it is apparent that the technique described so far can easily be repeated for the major alternatives (e.g., to build a new factory in place of a factory extension), it is not a satisfactory way of handling variability of the individual estimates used in the calculaton. What is required is a method similar to that sometimes used for network time estimates, where three time estimates, (optimistic, most likely and pessimistic) are quoted when uncertainty exists.

In specifying financial uncertainty however, a more flexible method is adopted whereby each alternative estimate is given a*probability* rating. Here the probability rating is an expression of the chances of that particular estimate becoming a reality. It can be likened to the odds quoted for horses in a race, which are the bookmaker's assessment of the chances of each particular horse winning (or rather not winning). The number of successful bookmakers gives some weight to the accuracy of the probability estimates.

In the case of project risk evaluation, the probability is usually expressed as a decimal or as a percentage, this showing the "weight" to be attached to that particular estimate. A series of calculations are then made to determine the profit which will result from alternative cash flow patterns. This calculation takes account of the "weight" applied to each cash flow value.

The method is to use a sequence of random numbers to guide the selection of dependent variables to be used. Each of the estimates is assigned a range of numbers the size of which is proportional to their probability. From a standard table of random numbers, the first number is taken and this is checked with the range assigned to each variable and when correspondence occurs that estimated value is taken for the calculation. The next random number selects the estimate for the next factor and so on until a full set is obtained upon which to perform the profitability calculation. When a set of data has been selected by this means, the calculation is made and the answer recorded. The calculation is repeated a number of times using figures obtained by the random number procedure described. After each calculation a count is made of the number of times that each profitability yield rate has occurred. The distribution of the answers (i.e. number of times each particular yield rate occurred) then gives an indication of the relative probability of obtaining each answer.

The calculation is, therefore, in the nature of a simulation of alternative solutions and when repeated for a sufficient number of times gives an assessment of yield probability. A sample of 100 calculations is perhaps the smallest quantity which will give a reliable indication and it will be apparent that the use of a computer is essential for this chore.

The answer appears as the frequencies with which particular profitability percentages are likely to occur over the full range. An example of the results from this calculation on the D.C.F. Yield of the "Blackmore Extension" (Figure 9) is shown in Figure 10. Here 500 calculations have been undertaken to find the D.C.F. Yield with a variety of input data having different probabilities. The number of times each percentage was calculated is shown alongside that particular percentage. It can be seen that 10% with 168 occurrences gives the highest expectation, but there is a small chance that the yield might be as high as 12% or as low as 5%. If all figures achieved fall within forecast ranges supplied, there is no chance that the profitability will fall outside the calculated range.

In Figure 10 these frequencies have been converted to percentages and plotted graphically as a line of XXs. This chart then gives a quick guide to the range and probability of particular project outcomes.

The computer has also plotted the cumulative percentage (shown as 0—0—0 in Figure 10). This is merely the sum of the frequency distributions converted to a cumulative percentage and provides a convenient reference chart, enabling the probabilities of achieving particular yields to be read off at a glance. For example, it can be seen that there is a 90% chance of achieving 8% or better and only a 10% chance of obtaining 11% or better. Conversely, there is an even chance (50% probability) that the actual D.C.F. Yield will fall above (or below) 9.6%.

With probability frequency distributions of the main factors involved in financial evaluation of a project, management have a useful way of analysing financial uncertainty. Risk can be categorized and so a degree

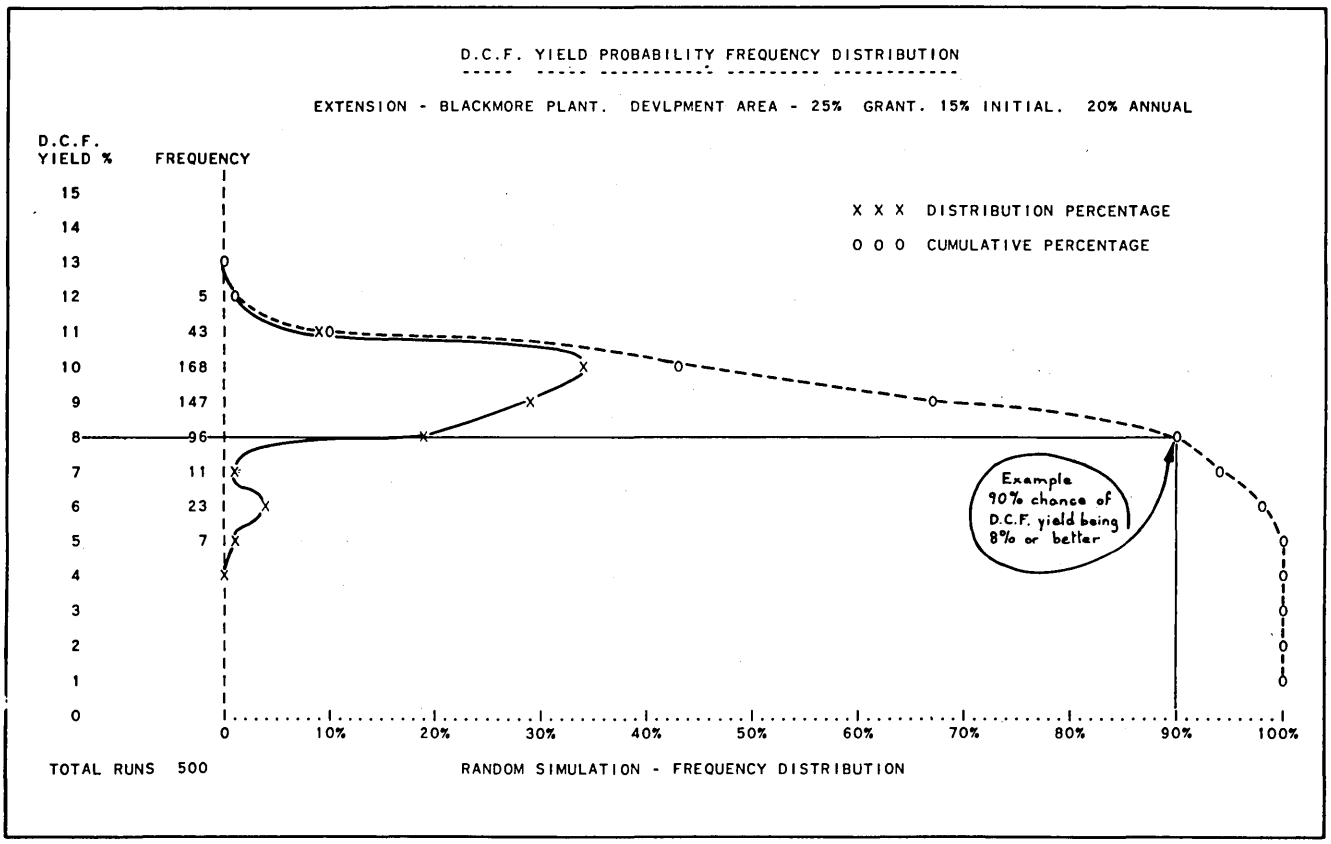


Figure 10—D.C.F. Yield probability frequency distribution

of segregation is achieved between those schemes which are viable propositions and those which are either distinctly unprofitable or involve an unacceptable degree of risk.

REFERENCES

1 H S WOODGATE
Planning by network
Business Publications Ltd London

2 International Computers and Tabulators Ltd 1900 series PERT and 1900 series PROP
The computer programs referred to are ICT Application Packages PERT (Programme Evaluation and Review Technique) and PROP (Profit Rating of Projects)

Winged words: varieties of computer applications to literature

by LOUIS T. MILIC
Columbia University
New York, New York

On August 6, 1961, an item appeared in the *New York Times* describing a rather unusual project that a young classical scholar named James T. McDonough had been actively pursuing. According to the account, for the previous four years McDonough had been reducing the Iliad of Homer to patterns representing the meter of the Greek epic, key-punching these patterns on a large number of punch cards and running them on an IBM 650 in the hope of discovering whether the uniformity of the patterns showed that the poem had been written by a single author, a question about which there had been a great deal of argument during the past century. McDonough hoped to earn a Ph.D. at Columbia with this work. Whether he succeeded or not, he had made a landmark in being one of the first who used computers in the solution of a literary problem. Since that beginning, others have availed themselves of electronic aid. As it is now ten years since the beginning of the relationship between computers and literature, it may be time to survey some of the results, to consider what has been achieved and what the prospects are.

In talking about literary computation, I want to confine myself to activities which are strictly literary. There are a number of related activities which border on literature but which are really tangential or preliminary to literary study. Among these I include machine translation, the making of concordances, attribution study, editing and bibliography, as well as most kinds of linguistic study. The reasons are relatively simple.

Machine translation arose out of a suggestion made in 1947 that machines might help to deal with the large bulk of foreign-language materials, especially in Russian, that government agencies and scientists found it necessary to go through in order to keep up with current developments. At first, owing to certain oversimplified ideas about the structure of languages, this seemed like a reasonable task to ask a computer to perform. Naturally the government was interested in furthering this research and invested substantial amounts in hardware and programs. The linguists were excited by the challenge and devised ever newer grammars to deal with the binary nature of the machines. Gradually, the hope of success dimmed as a realization of the incredible complexity of natural language gradually emerged. Not machine limitations but inadequate knowledge about the processes of the human mind and the nature of language finally doomed the machine translation project.

A year ago, the Automatic Language Processing Advisory Committee of the National Research Council issued a report in which it was concluded that, although machine-aided translation might be deserving of further support, pure machine translation could no longer be considered a practical possibility and ought not to receive further financing. The difficulties that machine translation programs have in distinguishing between literal and metaphoric uses of language and in dealing with idioms and problems of context are notorious. They can be illustrated by two examples, probably apocryphal. The expression "out of sight, out of mind" was supposed to be translated into Russian, where it became "invisible idiot." The phrase "the spirit is willing but the flesh is weak" returned from the Chinese analyzer as "whisky O.K., meat no good."

Whether these are true or not, they suggest the superiority of the human mind in dealing easily with levels of literalness. The twenty years and the millions spent on the effort to develop translation programs were not wasted on a mere effort to inflate the human ego. Though little translation actually took place, a great deal was learned about the nature of language. It may be claimed that postwar linguistics was revolutionized by the discoveries of machine translators and their auxiliaries, the mechano-linguists.

The field of computational linguistics is very active now and a great deal of valuable research is taking place in it which will ultimately be of use in translation and even in literary analysis. Work in automatic syntactic analysis, sentence generation, semantics has implications

for all kinds of word-connected activity. It is, however, remote from the immediate concern of the literary scholar and is not properly included in literary computation. All these linguistic projects are related to the scientific study of language, which today has the status of a behavioral science. The study of language as the medium of literature is quite a different realm, however much it may overlap that of linguistics, because its main concern is not with the characteristics of the code itself but with the individual and aesthetic use of the resources of language.

For that reason another whole class of studies may be excluded from consideration here, although they seem to reflect a much nearer concern with literature. I refer to concordances, dictionaries, glossaries, indices verborum, word-lists and bibliographies. Computers have vastly facilitated the compilation of these tools, and these tools do have something to do with the study of literature, but the use of computers in literary research is something different from the mere construction of lexical works of reference, which have existed in one form or another for centuries and which, when completed, are not essentially different from their handmade predecessors. The essential use of computers in literary research should diverge both quantitatively and qualitatively from the conversion of manual to electronic processing of data. The computer has made possible the processing of information in such quantities that no man's lifetime or energy could previously have contained it.

The mere amount of computer processing is a kind of innovation that we owe to technology. The class of studies that best fits this description of quantitative innovation is that of attribution, which has developed considerable activity since the advent of computers. To attribute an anonymous or uncertain work to its author requires processing a substantial corpus of text for each possible author and comparing its features with those of the work in question. Previously such attributions were made impressionistically on the basis of intuitively-perceived similarities or differences which could only be vaguely described: "This poem or this essay *sounds* like the work of Pope or Shelley or Ruskin." The inherent characteristics of the computer have necessitated the formalization of aspects of style for electronic processing. The predominance of short sentences, or of certain types of function words, the presence of certain grammatical constructions or favored lexical items, intervals between successive conjunctions, statistical properties of sentences or word-length distribution are examples of formal features of style.

The resultant combination—large corpora of text and cmpirical features of description—has made possible the identification of disputed works in ways that could not previously be imagined. Those who conducted the attribution studies on *The Federalist* papers, the *Letters of Junius* and the Epistles of St. Paul dealt in millions of words and have lived to tell about it. In the process of providing descriptions of the text explicit enough for the machine, they have added to our knowledge of these texts. Naturally, their results have not found favor everywhere, but they are on sound ground statistically and they have in fact merely ratified prevailing opinions in all three cases. Doubtless many more such studies will be undertaken now until the supply of disputed works runs out. One may look forward if he wishes to a definitive settling of the Bacon-Shakespeare-Marlowe contest.

Despite their usefulness to students of late eighteenth century political writing and of New Testament Greek, these studies are not essentially literary either. Their main interest is historical. They answer the question: "*Who* wrote this?" The literary information produced is merely a byproduct of the investigation. To be sure, a succession of attribution studies would provide extremely valuable information—information of which we have but the outlines at present—about the historical development of the English language. Such information, however, is linguistic rather than literary in nature. It is the background for stylistic studies but it is not itself literary. It is related to but not a fundamental part of the basic literary questions, which underlie the vast mass of literary scholarship.

If we now look at what has occupied scholars during the first decade of literary computation, we may be able to say whether they have been concerned with literature or with the preliminaries. Turning first to published work, we have some large projects resulting from the cooperation of a number of individuals and institutions. The Cornell Concordances, jointly fathered by Cornell and IBM, now cover Matthew Arnold, Emily Dickinson, and William Butler Yeats. The same group has plans for a number of additional works including most of the English poets whose works have not yet been so favored. A French group at Besancon has been conducting studies of the French vocabulary and making word-indexes of French poets and playwrights, publishing their results in two periodicals of their own. One of their separate publications is a concordance of Baudelaire which came out in the same year as one made by an individual scholar in this country. There is also a concordance to the Revised Standard Version of the Bible, which was published right at the beginning of our decade and a number covering medieval works in English and German which have just come out. It seems clear that for many scholars, using a computer has meant making a concordance.

Another large project was the attempt to solve the attribution problem in *The Federalist Papers*. Alexander

Hamilton and the editor of the *Papers,* James Madison, had long been considered in contention for the honor of having written a certain number of these pieces. Many historians were inclined to give them all to Madison despite the circumstantial evidence for Hamilton's claims to authorship. Two statisticians, one at Harvard and one at Chicago, decided to test the value of the Bayes theorem by applying it to this problem. With the help of a corps of assistants, two computers and a variety of government grants, Mosteller and Wallace concluded, as the scholars had done, that Madison had written them all. Because their concern was in statistics rather than in literature, their results do not have much interest for literary scholars. The work of a Swedish student of English literature, Alvar Ellegard, on a similar problem, the authorship of the *Junius* Letters, has been more interesting because of the information about the language of this period that he turned up. His conclusion about the authorship of the Letters coincided with prevailing opinion.

The researches of the Rev. Mr. Andrew Q. Morton of Scotland and his statistical colleagues is in a slightly different category. They have tried to distinguish between the various Epistles of St. Paul, the genuine and the spurious. Partly because of the manner in which his claims were presented and partly because of some sense among the public that the final sanctuary had been invaded by the machine, Mr. Morton has called down on his head the anger of a great number of people, including even some of the members of his cloth who are themselves using computers. Morton's results have not been fully made public, but he seems to have also found himself in agreement with previous, manually-assisted, scholars in his field. The criteria he used are not unlike those applied to *Junius* and the *Federalist*—frequency and distribution of function words in the text—but the text he uses is necessarily less reliable than theirs. After all, original copies of the eighteenth-century journals still survive but the text of St. Paul is in altogether a different state. The controversy continues to give off energy.

Questions which may be considered editorial were tackled by two scholars who used a similar technique in widely separated places. Both took advantage of the computer's ability to make a great many precise comparisons in trying to decide by means of spelling which text of Dryden or of Shakespeare had greater authority. In a sense the procedure is like that of the concordance maker with one important difference. A concordance program can only with difficulty be adjusted to recognize spelling variants of the same word. The studies just mentioned took advantage of this limitation in discovering spelling variation.

Projects even more remote from strictly literary work have been done and include a bibliographic index to the whole run of a Spanish literary journal, a million-word corpus of modern American English—this latter not published but stored on tape and available for consultation—and some collation and editing procedures that are of interest only for technical reasons.

Some very ambitious pilot studies have emanated from the workshops of Mrs. Sally Sedelow, now of Chapel Hill. Her interest, like my own, is in computational stylistics, the study of idiosyncratic patterns in individual writing. One of her programs converts specified verbal characteristics into graphic equivalents for easier comparison. Thus each noun and verb in a text could be indicated by a particular symbol, all other words being represented by zeroes. The noun-verb distribution would then be clearly visible and could then be evaluated. The trick of course is to think of the right things to look at, things that will tell us something about the text. The other program is more conventional, in the sense that it resembles a technique already in use for some time in the social sciences and named Content Analysis. The General Inquirer system, only recently applied to literary problems is a well-known example of a computer implementation of this technique. In essence, it consists of a thesaurus of themes and categories. If a text contains a sufficient selection of terms from a given category, it is concluded that the writer was concerned with that theme. Thus Mrs. Sedelow concludes from the number of words about lunacy (*mad, madly, madness, insane, disease*) in the first act of *Hamlet* that Shakespeare had this in mind when he wrote the play. Doubtless more esoteric conclusions can be reached by studying word-clusters and word-associations. At any rate this approach has the virtue of attacking the semantic component of language, which has been a great problem to all literary users of computers.

Not to overlook present company, I should also mention Professor Raben's well-known study of the influence of Milton on Shelley. In trying to pinpoint this debt, he tried to find how often in any sentence, Shelley used Milton's actual words. Contrary to his most optimistic estimates, he found an amazing number of such uses, clearly demonstrating the extent to which the later poet had incorporated into his mind the words of his predecessor. As might also have been expected, the handling of poems running into 200,000 words in the aggregate caused a certain number of space problems in the computer itself.

My own study of the style of Jonathan Swift, part of which was done on an IBM 1620, may perhaps be properly added to the end of this list, at least because it was only published this year though completed in 1963. My concern was to discover the individual features of this writer's style and to draw some literary conclusions

from this. After programming, the main technical problem I faced was the large amount of time that my list-processing procedures were using up.

As can be seen from this list, all but a few of these results of applying computers to literature have produced data sure to be useful to literary scholars—works preliminary to literary study—but are not themselves literary studies for the most part.

If we move now to work in progress as it is listed in the May issue of *Computers and the Humanities,* we find a vastly increased amount of activity. There are 120 projects listed under "Literature," though some scholars are responsible for more than one. Under examination, these break down into the following components. Predictably enough, the largest class (53) consists of concordances, dictionaries, word-lists, indexes, and catalogues of lexical items. The second largest category (25) includes various kinds of linguistic studies, programs for analyzing the linguistic characteristics of languages rather than of authors. There are seven bibliographical projects and six concerned with editing, collating, formatting and text history. Another six are devoted to various aspects of content and semantic analysis and the discovery of keywords. Five are attribution studies and another five are studies of meter and rhyme. Four are in machine translation. Of the remaining nine, two represent attempts to work up programs to serve literary scholars and are therefore really projects in information processing. This leaves seven which can be classified as strictly literary.

The descriptions provided are not full enough to permit complete understanding but it is possible to hazard some guesses as to what these projects may attempt. Two are studies of individual writers, one on a psychological basis involving word or image clusters, the other through his syntax. There is an attempt to determine whether a sonnet style exists. A comparison between a book of proverbs and a play is intended to show the reliance of the dramatist on proverbial sayings. There is a census of the roles of actors during a certain period to determine the nature of their specialization. And there is a study of the relation of grammatical deviation to mental disturbance, a matter of some interest considering how many poets have been or have been considered crazy. Except for the emphasis on linguistics, the distribution is similar to the earlier one.

Anyone who was not aware of the computer implementation of these projects and compared them to those recorded in such a Bibliography as the one published annually by the Modern Language Association might reach the conclusion that a revolution in the study of literature had taken place. Nearly half of the projects devoted to making reference-lists, nearly a quarter to linguistics! To be sure, the two samples differ considerably in size. The current issue of the PMLA Bibliography, recording almost exclusively items published in 1966, contains more than 20,000 entries covering work on all the major European languages since the Middle Ages. In it there is a small sub-sub-section on Computer-Assisted Literary Research, which contains some forty items, some of them merely general or popular explanations. At most this activity represents a very small fraction of the admittedly excessive total: one-fifth of one per cent, or one literary scholar in 500 is working with computers.

Because there is no classification of the items by type in the *PMLA* Bibliography, it would be very time-consuming to draw up a table, similar to the one just presented, for the efforts of traditional scholars. A casual examination of a random 120 items reveals, however, a predominance of historical studies of texts and documents, related social and political investigations, explications and criticisms of individual works, as well as some stylistic and linguistic studies, probably based, as is generally the case, on inadequate data. Without question, a number of all these studies could have benefited considerably from the data-gathering and data processing power of computers. In fact, it is probable that some studies are of doubtful validity because of the unrepresentative nature of their database. Traditional literary scholarship is notorious for extrapolations that go vastly beyond the data and even for conclusions reached without primary data of any sort.

What this suggests about the relationship between traditional and computer-assisted literary research is that both kinds of scholars seem to be pursuing the same ends but that what I may perhaps call the "modern" scholar has in the main limited his scholarship to certain kinds of preliminary work which is the basis for conclusions of a more far-ranging character. Concordances and the like permit studies of works and authors to be more soundly based. Attribution studies enable the critic to feel more positive about the canon of an author's work. But all these studies ultimately serve the same master. To be meaningful they must stand in a certain relation to the basic critical questions which determine the nature of any art.

What are these literary questions to which such deference must be paid? They are all primarily founded on the aesthetic aspect of human activity, the third member of the Platonic trinity of the good, the true and the beautiful. More specifically, literary criticism and scholarship must concern themselves with distinguishing between the aesthetic and the everyday, good literature and bad, poetry and mere verse. In so

doing, the scholar must give his attention to the nature of the aesthetic effect, the creative activity of the writer as opposed to the merely routine aspect of communication. This question was, until recently, unique with literature because its practitioners use the same language in writing odes and sonnets as is used in the daily newspaper, the freshman theme and manuals of instruction for computers. Pop art (the conversion of tomato soup cans and giant hamburgers into the substance of art), the underground film (the 8-hour showing of a man sleeping), and certain tendencies in music (the bizarre use of musical silences and ugly sounds), have, however, eroded the uniqueness of this feature peculiar to literature. These other arts have now been compelled to take a stand on the basic aesthetic question "What is art?" before being able to arrive at the next one, "What is good art?"

For literary students, the basic question remains "What is literature?" In the process of trying to answer it, the scholar finds himself dealing with a variety of subordinate questions, the answers to which he hopes will lead him to solve the main one. A favored form of the basic question about literature is "What is the meaning of this play, this novel, this lyric poem?" This question branches out into other questions of meaning: of words, phrases, themes, plots, symbols, stylistic devices . . . . Questions of meaning are, as the linguistic philosophers have shown and as everyone now knows, very difficult to answer. In part this is because the verification of problems of meaning is not empirically possible, as meaning is an abstraction at a certain remove from the events under examination. Disagreements about meaning, about the interpretation of literary works, abound in literary study. In a sense every interpretation is correct or at least justified since it may be supported by a proper selection of evidence and since there is no established priority governing the evaluation of evidence. Thus literary interpretation generally depends for its effect on the persuasiveness with which the selection of evidence is presented and it usually relies for its acceptance on a certain set of beliefs or expectations common to the interpreter and his audience. For example, the question "What does *Hamlet* mean?" has been answered in this century by reference to the possible incestuous relationship between Hamlet and Gertrude and Hamlet's supposed Oedipus complex, which in turn have been traced to some emotional difficulties in the playwright. But other meanings of Hamlet have been successfully defended which are supported by a different selection of evidence.

As is well known, there can be no progress in interpretation. Explications survive for as long as the willingness to believe the theory on which they are founded persists. When the winds of critical doctrine change, what was previously acceptable—theory, interpretation, evidence—is swept away to be replaced by the newer thing. This is a discouraging state of affairs but one to which literary scholars have become adjusted. Their way of adjusting to this fluid and unstable situation is by the reduction of big problems to little ones, by the conversion of *why* questions to *how* and *what* questions.

Preliminary to any inquiry about the meaning of a given work is usually a set of subordinate questions, some of which may seem rather remote from the main event. Thus, the study of *Hamlet* implies the study of the medieval theatre, beliefs about lunacy, ghosts and family relationships, the sources of this particular play, the shape and appointments of the Elizabethan playhouse, Shakespeare's life, his philosophy as it is reflected in the speeches of his characters and in his imagery, his language as it differs from or corresponds to the language of the playwrights and writers of his time, and an innumerable list of sub-questions. Presumably, when all the evidence is in on these lower-echelon matters, the main question—"What is the meaning of *Hamlet*?"—can be tackled, unless someone comes along with a critical theory that denies the possibility that plays or other literary artifacts can have meaning, apart from mere existence. The words of a modern poet record this position of critical nihilism: "A poem should not mean but be."

Without an unceasing concern for the ultimate necessities, the questions of meaning and value, any study is in danger of becoming merely the trivial sorting of artifacts, the solving of puzzles or riddles no more significant than a newspaper crossword. In other words, literary scholarship, computerized or traditional, must be informed by this concern for what literature is and means and for the things that literature springs from and tries to illuminate. Computer scholars are more vulnerable to such a danger than traditional scholars because the traditional tools of research—cards, files, pencils and typewriters—do not exercise the dangerous and autonomous fascination that the electronic data-processors do. The computer study of literature always threatens to take over the scholar, who becomes seduced by the ease with which it can do certain things into abandoning his real goals and responsibilities. At the same time, he is subject to another sort of accusation which is quite the opposite. If he uses a computer, he is expected to solve all the outstanding problems of literature, simply because the real accomplishments of the computer and the efforts of the manufacturers' public relations men have accustomed the public to expect decisions, solutions and miracles from the computer, as a matter of routine. Anything less counts as a failure. These and other

jeopardies face the literary scholar who has turned for help to a computer.

The problem is truly paradoxical. The traditional literary scholar cannot solve the great problems unless he first solves the small ones. These invariably consist of the accumulation and compilation of data, minute in size and immense in quantity. If he immerses himself in these, he is very likely to lose sight of his original purpose. If he does not, his conclusions are mere baseless speculations. To this paradox, the computer can bring a solution because of its ability to undertake the drudgery required for answering the subordinate questions.

Nonetheless, it is difficult to escape the conclusion that computer-assisted literary scholarship has until now been woefully conservative. It has done little to exploit the machine's genuine possibilities for qualitative innovation. It has largely limited itself to the mere quantitative aspect. These efforts will doubtless earn some praise as the results become available and useful to the community of scholars, but they will not inspire other scholars to emulation because the results are not truly inspiring or exciting. Not until the computer scholar turns out results which diverge sharply from what has been done before will he earn the respect and interest of his traditional colleagues.

One explanation of the conservative nature of the computer-assisted projects has to do with the relationship between the scholar and the machine. He has learned to think of it as a highly efficient but brainless clerk—despite everything written about artificial intelligence. Therefore he has entrusted it with merely clerical tasks. Moreover, he has usually employed an intermediary to convey his instructions because he is not himself sufficiently conversant with its language to do so himself. In a sense, therefore, he is doubly dependent and doubly limited: he has a foreshortened view of the computer's abilities and he must depend on the understanding of another person to express his needs. He must free himself of both these limitations if he wishes to make his scholarship creative. Both, it seems to me will yield to the one cure: the scholar must learn to be his own programmer. That is axiomatic. He cannot learn what the computer can do if he has to ask another to interpret for him. With the development of new high-level languages like SNOBOL, competence in which can be acquired even by the stiff reflexes of the middle-aged scholar, though not without effort, there can be no excuse for remaining technologically illiterate. The scholar who familiarizes himself with the means of communicating with his computer will learn at the same time how extensive are its possibilities, how untried its opportunities.

The aims of humanistic scholarship, according to the recent words of a well-known classicist, should be primarily educational. They should, that is, instruct the scholar and enlighten his instruction, especially in the sense that a knowledge of the past can help one to judge the present. If the fulfillment of the literary humanist lies in this sort of activity, the computer properly used can make a considerable contribution.

Music and computing: the present situation*

by ALLEN FORTE
Massachusetts Institute of Technology
Cambridge, Massachusetts

Perspective

There may be those who find that the terms music and computing form an unlikely pair. We remind them that the monochord, a device usually associated with music, was one of the first scientific measuring instruments. We need not stop there. In virtually any historical period one finds an interaction between music and science and mathematics. With respect to the seventeenth century, for example, Claude Palisca has observed:

> *In any discussion of science in the seventeenth century, among the names that inevitably arise are those of Galileo Galilei, Marin Mersenne, René Descartes, Johannes Kepler, and Christian Huyghens. It is no mere coincidence that these . . . were all trained musicians and authors on musical subjects . . . because music until the seventeenth century was a branch of science and held a place among the four mathematical disciplines of the quadrivium beside arithmetic, geometry, and astronomy.*[1]

The interaction has not been uncontroversial. In the fourth century B.C. Aristoxenus, one of Aristotle's most eminent pupils, took issue with the Pythagoreans, who maintained that the science of harmonics, regarded as central to music, was based upon numerical relations. Aristoxenus asserted that a rational interpretation must take into account the more basic factors of sense-perception and memory. This view was echoed by the celebrated mathematician and encyclopedist, D'Alembert, in the preface to his treatise on Rameau's theory of music:

> *One can consider music either as an art, the purpose of which is to provide one of the principal pleasures of the senses, or as a science by which that art is reduced to principles.*[2]

A nineteenth-century scientist took a simpler view (and one perhaps substantiated by the current commercial musical product):

> *I conclude that musical notes and rhythms were first acquired by the male or female progenitors of mankind for the sake of charming the opposite sex.*[3]

Although the association has sometimes been ludicrous or even fraudulent, the point is that some aspects of music have long been involved with science, mathematics, and technology in some way. It is not at all strange, therefore, that a significant segment of contemporary work in music theory and composition should be concerned with logic, mathematics, and machines. Indeed, a natural synthesis of these seemingly divergent enterprises is even now taking place in computer-implemented music research and composition.[4]

This trend is especially evident in recent work in this country, much of which has been inspired by the theoretical formulations of Milton Babbitt[5] and set in motion by the pioneering efforts of Lejaren Hiller and his associates.[6] It is now evident that an intellectual climate exists—albeit in a very small group—such that one can predict with some degree of certainty that computer-implemented music research and composition will continue to extend and will produce significant results.

Because of the diverse applications that have been made or proposed, it would be pretentious to imply that a comprehensive view of the present situation can be given here. Nonetheless, an attempt will be made to indicate those directions and activities that are currently visible. Much of the work is long-range and experimental. Accordingly, it is necessary to say, once and for all, that we are still in a pioneer stage.

Sound-generation by computer

Sound-generation by computer involves the computa-

*This is a summary paper. The author's own work, to which passing reference is made, was supported (in part) by Project MAC, an M.I.T. Research Project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

tion of waveform samples at a specified sampling frequency and the conversion of these to an output waveform through a digital-analog device, a low-pass filter, and an amplifier.[7] For "real time" generation, which presumably is to be preferred to the "off-line" situation where digital computer output is converted to analog form by a transducer separate from the computer, speed and storage capacity are critical factors. It appears that more effective use of the computer for sound-generation must await hardware improvements, since the "off-line" situation requires an excessive amount of machine time for the production of complex sounds, while real time generation necessitates compromises in the form of limitations upon obtainable sounds.

At present, the generation of sound by computer affords the researcher and composer at least three interesting possibilities: (1) experiments in auditory perception—characterized by Babbitt as "the most refractory of areas";[5] (2) the study of musical "grammars"; (3) the development of original compositions. The greatest amount of activity has occurred in the latter category. Early credits go to Max Mathews and the MUSIC IV program for sound-generation and to Lejaren Hiller at the University of Illinois. More recently, computer generation has been carried out at Princeton (by James Randall,[8] Godfrey Winham, Hubert Howe, and others), at Yale (by James Tenney), and at M.I.T. (by Ercolino Ferretti[9] and by A. Wayne Slawson).

The second category (study of musical grammars) is relevant to contemporary work in music theory. Is it possible to write a computer program to produce new music in a familiar style? The serious issues and problems involved here have been obscured, unfortunately, by the occasional efforts—dutifully recorded by the press—to produce music that "sounds like" Mozart, Bach, Irving Berlin, etc. To my knowledge, no such effort has been successful. (It should be remarked, in this connection, that actual sonic output would not be required. For example, the computer might produce or display a score in complete music notation.)

Music research

Under this heading come a variety of applications: information retrieval, style analysis, study of musical systems, and the development of music representations for computer processing. (Most of the projects cited below are described in more detail in the compilation by Edmund A. Bowles listed in the references.[10])

In the information retrieval category perhaps the most impressive project is the important RILM (Répertoire International de la Littérature Musicale), directed by Barry S. Brook, that is being carried out at New York University Institute for Computer Research in the Humanities.[11] The long-range goal of this project is bibliographic control of the scholarly information about music past and present.

Several eminent scholars are interested in pattern-recognition, with a view to codifying style-characteristics for a particular corpus of music. Among these are Arthur Mendel (the vocal works of J. S. Bach), Lewis Lockwood (the masses of Josquin), Jan LaRue (Haydn symphonies), and Harry Lincoln (Frottole repertory). This work is characterized in part by what might be called "overlay" procedures, the comparison of variant texts for relevant similarities and differences, and thus has an affinity to certain work being done in literary research.

Research in what may be called, loosely, musical systems has been undertaken by Stefan Bauer-Mengelberg and Melvin Ferentz,[12] by Michael Kassler,[13] Hubert Howe,[14] and by the present writer.[15] These projects are characterized by a concern with combinatorial problems, complex decision structures, and non-statistical mathematical models.[16]

Both in style analysis and in the study of musical systems a distinction can be drawn between "numeric" and "non-numeric" processing. If the researcher deals indirectly with music, that is, if his data consist of numeric sets representing some musical property or properties, information-loss is assumed and the problem is usually solved in a straightforward way, using available mathematics. If he deals more directly with music, however, the question of input data and information-loss becomes central. What is to be the object of study? Bauer-Mengelberg[17] maintains that it is the score and makes a cogent case for a syntactic representation that is complete for any composition. Other researchers take a more casual view and are content with incomplete or *ad hoc* representations. The issue is interesting and significant, for it may ultimately affect the viability of a research project.

The question of music representation is but one of many we are beginning to cope with. Interpretation of output, which involves criteria of parsimony and significance, formalization, and the development of efficient algorithms is a matter of immediate concern. The question of appropriate high-level languages and their natural data-structures is also in the foreground for those of us who are active in computer-implemented research.

A look ahead

We confine our remarks here to music research, although some of them are probably relevant to composition as well.

To a large extent the future of music research vis-a-vis the computer depends upon education. If computer-implemented research is to have a significant effect it must be undertaken by more scholars than are now active in

that area. They must be scholars of the highest rank, well versed in subject matter and sufficiently competent to be able to program fluently and to supervise programming, where that is desirable and feasible. It might be noted here that the Harpur Seminar on Music Research and the Computer, a two-week summer course directed by Professor Harry Lincoln, represents a pioneer effort to effect a rapprochement of music scholar and computer.[18] For younger scholars—mature graduate students in particular—the problem is simpler. They must be taught to use the computer as a normal part of their formal education. This is especially important for those in the area of music theory, but is probably also essential for scholars whose main interests are historical—on the assumption that even though the scholar's historical interests might be confined to the period from 1601 to 1603 he lives in the 20th century and presumably should have access to contemporary research facilities.

It is wise to remember that many problems in music are complex. The extent to which data-processing technology will render solutions more accessible than do traditional procedures remains an open question. For example, it now appears that computer-generated graphic displays offer new resources for the editing procedures that are central to much work in musicology. Yet, a great deal of work must be done before machine-implemented editing can cope with the notational systems of various historical periods or with such a complex representation of the human creative process as a page from a Beethoven sketchbook. In attempting to cope with such problems, however, we can expect that traditional music scholarship will obtain insights that may determine extensive critical revisions of conventional methods and criteria.

REFERENCES

1 C V PALISCA
Scientific empiricism in musical thought
In Seventeenth Century Science and the Arts edited by H H Rhys Princeton University Press 1961

2 J D'ALEMBERT
Élémens de musique
London 1772

3 C DARWIN
The descent of man
London 1872

4 A FORTE
Computer-implemented analysis of musical structure
In: Papers from the West Virginia University Conference on Computer Applications in Music West Virginia University Library Morgantown 1967

5 M BABBITT
The use of computers in musicological research
Perspectives of New Music vol 3 no 2 Spring-Summer 1965

6 L A HILLER L M ISAACSON
Experimental music; composition with an electronic computer
McGraw-Hill Book Co New York 1959

7 JAMES C TENNEY
Sound generation by means of a digital computer
Journal of Music Theory vol 7 no 1 Spring 1963

8 J K RANDALL
A report from Princeton
Perspectives of New Music vol 3 no 2 Spring-Summer 1965

9 E FERRETTI
The computer as a tool for the creative musician
In Computers for the Humanities?; A Record of the Conference Sponsored by Yale University on a Grant from IBM January 22-23, 1965 Yale University Press New Haven 1965

10 E A BOWLES comp
Computerized research in the humanities; a survey
ACLS Newsletter, Special Supplement June 1966

11 B S BROOK
RILM répertoire international de la littérature musicale
Computers and the Humanities vol 1 no 3 Jan 1967

12 S BAUER-MENGELBERG M FERENTZ
On eleven-interval twelve-tone rows
Perspective of New Music vol 3, no 2 Spring-Summer 1965

13 M KASSLER
A sketch of the use of formalized languages for the assertion of music
Perspectives of New Music vol 1 no 2 Spring-Summer 1963

14 H S HOWE
Some combinational properties of pitch structures
Perspectives of New Music vol 4 no 1 Fall-Winter 1965

15 A FORTE
A program for the analytic reading of scores
Journal of Music Theory vol 10 no 2 Winter 1966

16 J ROTHGEB
Some uses of mathematical concepts in theories of music
Journal of Music Theory vol 10 no 2 Winter 1966

17 S BAUER-MENGELBERG
The truth, the whole truth, and nothing but the truth
Paper read at The Computer and Research in the Humanities a conference held at the University of North Carolina Chapel Hill March 1967

18 J PRUETT
The Harpur College music-computer seminar: a report
Computers and the Humanities vol 1 no 2 Nov 1966

Computer applications in archaeology*

by GEORGE L. COWGILL
Brandeis University
Waltham, Massachusetts

INTRODUCTION

In preparing this paper I have tried to give an accurate general picture of the kinds of things which have been done with computers by archaeologists and to give some of my own views about things which most need doing, but I have not tried to list every archaeological application of a computer ever made. The emphasis is mostly on work done in North America, and I believe that the coverage of significant work in this region is quite complete. I also include a good deal of what has been done in western and central Europe, but much less on eastern Europe and the Soviet Union. Gardin[1] has recently cited several important Russian publications which so far as I know are not available in English.

Archaeological use of computers is still in a very early stage. Several archaeologists had used machine-sorted punched cards much earlier, but the first archaeological applications of electronic data processing I know of were by Peter Ihm and by Gardin and Garelli in France around 1958 or 1959.[2] The earliest use I know of in this country was by James Deetz in 1960.[3] Subsequently there have been about 5 or 10 major projects involving computers and archaeology in North America, about the same number in the rest of the world, and many smaller ones. These projects have covered a wide range of quality and sophistication. As could be expected, in a few cases there was a too-hasty effort to use a machine because of the attraction of something so fashionable and novel. We have had our share of people who, in spite of all disclaimers, seem really to have expected that marvelous results might follow easily once their data were somehow "computerized." Naturally nothing really marvelous has come out of these early studies. Results have always been interesting, and in some cases important contributions to archaeological problems have been made. Yet, nothing done so far has convinced the archeological profession as a whole that there are any often-encountered tasks or problems for which computers ought to be used as a matter of course; that there are tasks for which it would show incompetence *not* to use a computer. My impression is that the majority of archaeologists are still watching results of computer work with attitudes ranging from hostility to friendly interest, and are not going to make any real commitment either to learning or to using machine techniques until there is more evidence that computers can really offer economies in the performance of familiar tasks, or that the results of novel computer approaches are really valid and intelligible. The incidence of intelligent comprehension of computers is still depressingly low among archaeologists of all ages, although it may be rising rapidly in the current generation of students. I think archaeologists really engaged in computer work have reached a "second generation" stage where it is more fully appreciated that a great deal of hard work, hard thinking, and trial and error are still needed before we can make the best uses (and non-uses) of computers, but nonetheless a stage where we have a substantial body of earlier efforts whose successes and failures we can learn from. It seems wasteful for either archaeologists or computer people trying to help archaeologists to begin computer projects today without knowing what has already been done or attempted in archaeology. An extremely important source of information on this work is the *Newsletter of Computer Archaeology*.

Important tasks

Most archaeological applications of computers fall

*The present paper has profited from comments on an earlier version made by R. A. Benfer, Junius Bird, R. G. Chenhall, Keith Dixon, J. C. Gardin, Dee F. Green, James Hill, F. R. Hodson, T. Hunter, Jesse D. Jennings, William Lipe, I. Scollar, A. C. Spaulding, S. Sedelow, and Bruce Warren. They have informed me of some new work, made some stylistic suggestions, and indicated places where my meaning was unclear. I gratefully acknowledge their help.

under the broad headings of data storage and retrieval or of multivariate statistical analyses. The two most important tasks we face with respect to these applications are developing optimum procedures for coding our data, and getting a better control of mathematics. Both involve much that can be done only by the archaeologists themselves; for only the people working with the data can say what kinds of things may be important to record about it, and certainly no one else can learn our math for us. This last point needs some stressing because archaeologists rarely know anything beyond high school algebra, and most do not know even that really well. My impression is that most archaeologists without computer experience believe that any competent programmer is also an all-round mathematical authority and do not realize that they may need important statistical advice which a programmer cannot give them. Probably programmers for their part often assume that archaeologists know more than they in fact do about the best ways to formulate or to solve their problems. Also, archaeologists who have gotten some kind of statistical answer out of a machine have been at best shaky about what to make of it once they had it, and in some cases have been confused or misled by the results. Whatever other roles computers come to play in archaeology, we will not be able to get very much out of statistical methods until there is a higher level of mathematical competence among archaeologists. Minimal requirements are that archaeologists be able to judge whether a suggested technique is really appropriate for their data and problems, and that they recognize the need for competent statistical advice as something distinct from programming assistance. It is also desirable that some one develop new, more suitable mathematical techniques in cases where nothing now existing is really quite what we want.

Archaeological data codes

The kinds of codes we especially need to develop are those for describing physical objects, both for data storage and retrieval and for statistical analyses. This is particularly so for broad classes of materials such as pottery, chipped stone, ground or polished stone, metals, and fibers and textiles. It can also be useful to encode patterned concatenations of objects, for example structures or burial arrangements. For any specific study, the most efficient coding will depend on the range of variations shown in a specific body of data. But for each of the broad classes of materials mentioned above, there are substantial universal similarities in what it is relevant to describe and in the limits of variation. It is certainly undesirable that every computer project develop a completely *ad hoc* coding scheme. This is not to say that we should not continue to explore and experiment with new approaches to old coding problems, nor that special codes will not be desirable for special data or special problems. Above all I do not mean that we should commit ourselves to some kind of "least common denominator" framework which will lend itself mainly to coding features which occur widely in many different cultural traditions. On the contrary, a major advantage to computer techniques should be a greatly improved ability to deal with fine distinctions in detail relevant only for the quite specific and localized entities which Lipe[5] calls "micro-traditions." The point is that there are kinds of features of near-universal importance which ought to be dealt with well by our codes, whatever else in addition they handle. Also, there are recurrent problems in expressing the way in which parts of an object relate to one another to form the whole object (what, to use the linguistic kind of approach favored by Gardin, might be called "syntactic" relationships). People starting new computer projects will very likely find that many of the problems they face have had to be faced in earlier projects, and they ought to consider earlier attempted solutions carefully and adopt a consistent scheme unless they can think of something to try which offers a possibility of working better. Clearly we should continue to try out many new things just to see how they will work, but we do not need to re-solve already solved problems or to duplicate old mistakes. General frameworks and standardized practices will save wasted effort and will help toward making the best balances between particularity and universality in specific codes; without requiring that codes for every data file for a given kind of material be highly similar. Important recent general discussions of coding include those by Gardin[1] and by Chenhall.[6]

Archaeologists are currently debating among themselves about the degree to which the features we find important for establishing categories of artifacts, and the categories themselves, could ever be demonstrated to be close to those really "in the minds" of the dead makers and users of those objects. Whether or not this will ever be possible, at least we should not put unnecessary obstacles in the path toward such a goal. Furthermore, for a wide variety of statistical techniques, we need to be able to generate numbers from the basic data file which will be reasonable expressions of the resemblance to one another of objects or of sets of objects; based either on all their recorded features or on some definite subset of their features. Both of these considerations imply that descriptive codes should always be reasonably related to human judgments made by workers experienced with the corpus of objects. Features judged to be quite similar should have this similarity reflected in their coding, and similarities in "syntactic" arrangements of features on objects should

also be reflected, even though the specific features may differ. Emphatically, this does not mean that we should merely find ways of translating currently popular descriptions into a code readable by machine, still less does it mean we should arrive at the same categories as before. Current archaeological description practice is to omit much detail because it is simply unmanageable, and very often categories are quite palpably derived by selective emphasis on certain features at the expense of others, or by methods which frankly strike other workers as idiosyncratic. I also do not mean to say that logical clarity guarantees good results, for it is easy to do something which is logically rigorous and consistent and absurd. We need a fine interplay between human insight and common sense and the vast literal-minded idiotic power of the machines. All this may seem to be stating the obvious, but I mention it, especially the need that the original data file use a code related to our perceptions about similarities in features and similarities in configurations of features, because it is quite possible to use codes which lack these properties. For example, one could easily store outlines of objects by listing a series of grid coordinates, but such a system makes it hard to emphasize differences at certain key points of the profile, or to express the strong fundamental similarities in objects which differ only moderately in some proportions.

There may be important lessons for us in the work being done on pattern-recognition by machine, but so far I know of no useful archaeological applications of these techniques. To describe the geometrical aspects of solid objects like pottery, stone, or metal, what I think we need are some important size measurements, locations of often-relevant "landmark" points, information about profiles at and between these points, symmetry properties, and information about location and character of special attachments to or deletions from the main body. Important non-geometrical kinds of information include raw materials, clues about methods of manufacture, clues about use, features of finish and decoration, context in which the object was found, and catalog number or other positive identification of the object. Chenhall has discussed these matters in an unpublished M. A. thesis,[7] although unfortunately his published work[6] does not treat coding shapes except in very general categories.

For geometric description of pottery vessels, I think the broad outlines of a good approach are clear. This is most fully described by A. O. Shepard.[8] McGimsey and Green,[9] and Gardin[1] illustrate very similar approaches. What is needed here is not some drastic revision, but improvement and standardization of this general method, insuring that it retains an "open-ended" quality which will permit its application to the details of any conceivable ceramic tradition. I am less in touch with work being done on chipped stone tools, but I believe that there is fair agreement about what kinds of things it is important to notice and less agreement on a standard way to encode these observations. Important work here includes the lithic typology conference reported by Krieger[10] and Weyer,[11] and the codes described by Binford[12] and Jennings.[13] Stern[14] gives a code for ground stone. Christophe and Deshayes[15] present a code for metal tools and weapons. An unpublished code for computer analysis of textiles, developed at the American Museum of Natural History, is discussed by Bird.[16]

The matter of developing general codes for decorations of objects, where the concern is with design elements, style, iconographic content, or subject matter, seems far more difficult. It is an area where methods adapted from descriptive linguistics seem very promising. Gardin[1] gives a very important discussion and illustration of some of these methods. Also very important, though not intended for immediate use with computers, is the work of John Rowe and his students at Berkeley,[17] and of Muller.[18] This is plainly a topic where archaeology and art history have many similar needs and problems.

One application of archaeological data codes which is somewhat distinct from their use in specific research projects is in the "computerization" of the catalogs of large museums. This task is under way or seriously projected by Dee Green at the University of Missouri and by Jaime Litvak and Felicity Thomas at the Instituto Nacional de Antropologia e Historia in Mexico City. Irwin Scollar [19] reports that this may be done at the Rheinisches Landesmuseum in Bonn.

Statistical studies

One reason why computers have not yet had any great impact on archaeological practice is that no one has yet completed and made available a file containing any really large body of important data coded to include what is relevant for important problems. Probably the largest published file is by Christophe and Deshayes,[15] which includes about 4000 metal objects, using optical coincidence cards rather than electronic equipment. The statistical studies which have been done so far have never been based on samples of more than a few thousand objects (at most, a few tens of thousands if very fragmentary objects or workshop debris are included). In many cases, of course, good samples of this order of magnitude are quite sufficient to produce important results, but it does mean that the volume of data processed has never been very large relative to the millions of objects (mostly small pottery fragments) which major excavations often produce. The im-

portance of computer studies will increase greatly when larger files of significant data are accumulated, especially as whole regions and substantial time spans come to be covered with some adequacy. Even so, it does not seem profitable to try to include all data on all objects excavated, and good statistical sampling design is a matter of increasing concern.[20]

Some important examples of archaeological applications of statistical techniques by computer include theuse of chi-square and regression by Freeman and Brown;[21] multiple regression by McPherron[22] and by Longacre;[23] factor analysis by Jennings,[24] Binford and Binford,[25] Cowgill,[26] Hill,[27] and Benfer,[28] proximity analysis by Hodson and others;[29] scalogram methods by Eliseeff;[30] and automatic classification methods by Hodson[29] and De La Vega.[31]

Few archaeologists have been involved in development of new statistical programs for their work. Deetz's[3] work in 1960 is one exception. For a number of reasons, his work was probably more influential than any other single project in persuading American archaeologists that computers might possibly be valuable for them. Deetz addressed himself to an original and important problem; whether the historically documented breakdown of social organization (particularly a pattern of matritocal residence) under increasing European pressures on an Indian village in South Dakota in the 18th century might be reflected in a parallel breakdown of clustering in ceramic design elements. But it is evident that no one with basic statistical competence gave this work any serious attention. Many archaeologists, notably Deetz himself, are well aware that even well-demonstrated changes in clustering of artifact design elements may not have clearcut social implications. What needs to be emphasized is that Deetz's demonstration method itself involved computations that were unnecessarily tedious and unnecessarily ambiguous, and should not be used as a model for further work. He attempted to assess degrees of association among crosstabulated attributes, by a technique which was ingenious but less useful than standard measures like phi or lambda.[32]

Kuzara, Mead, and Dixon[33] have developed what seems to be a very good program for the task known to archaeologists as "seriation"—arranging a set of units in the order which best satisfies the requirement that the more similar any two units are, the closer to one another they are in the final sequence. Archaeologists have had a fair amount of experience in doing this directly by manual rearrangement of the units, so there is already fair understanding of the rationale, and the convenience of doing it by computer is appreciated. Kuzara, Mead, and Dixon's program appears to work better than an earlier one designed for this purpose by the Aschers.[34] It has already been applied by other archaeologists, including a study of stone tools in Texas by LeRoy Johnson,[35] and it has good prospects of becoming quite popular. Its only serious limitation is that it amounts to ordering units along some one best axis or factor. Where there is any reason to think two or more factors may be relevant, it would be preferable to use a multidimensional technique such as factor analysis or something like R. Shepard's proximity analysis.[36]

In most multivariate approaches a pervasive theme is the drive toward certain kinds of parsimony. What are the best variables for discriminating between members of several categories, what are the best predictor variables for some set of criterion variables, or what are the fewest independent factors which account for most non-random variance in some larger set of variables? In many archaeological problems these are indeed the kinds of parsimony we want. Often, though, we really want something else; namely, the most parsimonious account of the patterning of *all* variables of some set. Probably we should rely less on methods developed for the reduction of experimental data (especially by psychologists, agronomists, and biologists), and more on analogies with descriptive grammars. Lounsbury's[37] approach to formal accounts of systems of kinship terminology is an especially important and lucid exposition of this approach. Excellent archaeological work along these lines has been done by John Rowe and those influenced by him at Berkeley,[17] working without computers. Rowe and his students have produced results far more important than anything which has been done so far in archaeology by computer, largely because they have applied a good method to rich bodies of data, while computer studies have been short on one or both of these scores. Muller[18] has also done important work in applying a generational grammar approach to a prehistoric art style. It is likely that computers could be used to make the "grammatical" approach more powerful and less laborrious, but this will require more hard and original thinking, than is demanded by the adoption of ready-made multivariate programs. Gardin's[1] work on codes for iconography is an important contribution in this direction. Sackett's[38] non-computer work on multiple contingency tables is superficially quite different, but is probably also leading in the same direction.

The general field of mathematical geography, or mathematical analysis of spatially distributed data, is another promising field for computer applications to archaeological data. Work here includes Lipe and Huntington's use of centrographic techniques for demonstrating differences in distribution of ceramic categories,[39] and the use of linear spatial filtering to im-

prove contrast in plots of magnetometer survey data by Scollar and Krückeberg.[40] Data smoothing and trend surface fitting techniques are also likely to prove useful. I am currently engaged in analysis of data from Teotihuacan, a 25 square kilometer prehistoric metropolis in central Mexico,[41] where differences between districts within the city is one major concern. We are working with a data matrix of 391 observations on each of possibly 4000 units. While most of our computer work utilizes multivariate statistical methods, we have also found it useful to produce maps by computer of data distributions using a program (SYMAP) developed under the direction of Howard T. Fisher, of the Laboratory for Computer Graphics of Harvard University. By far the greatest advantage over hand methods comes when functions of data at two or more points must be computed, as in smoothing or filtering procedures.

Other applications

One special field of computer work is on decipherment of ancient writing systems. An early attempt to decipher Maya hieroglyphs by Evreinev, Kosarev, and Ustinov at Novosibirsk was unsuccessful and strongly criticized by others, including Knorozov.[42] Current work in Mexico on a concordance of Mayan inscriptions is not aimed toward instant decipherment and is likely to be far more useful.[43] At least two computer projects involving Minoan writing are presently under way.[44]

A KWIC index of *American Antiquity,* a major American journal, has been produced by Dee F. Green[45] but is not yet published. According to Irwin Scollar[19] the annual and cumulative indexes of the *Bonner Jahrbuch* and a concordance of aerial photos of archaeological sites are all being compiled by computer at the Rheinisches Landesmuseum, Bonn.

Perhaps the most unusual computer application so far in archaeology is in connection with the work of G. Hawkins, who used computed ancient stellar positions for his study of the astronomical significance of Stonehenge. James Dow has also used this program for research on possible stellar bases for orientations of ancient cities and temples in Mexico.[46] Undoubtedly many more special applications of computers in archaeology will appear, in addition to their major uses for data storage and retrieval and for statistical and formal analysis of data.

REFERENCES

1 J C GARDIN
Methods for the descriptive analysis of archaeological material
American Antiquity 32 13-30 1967

2 J C GARDIN
Reconstructing an economic network in the ancient East with the aid of a computer
In Hymes D The Use of Computers in Anthropology Mouton & Co The Hague pp 377-391 1965

3 J DEETZ
The dynamics of stylistic change in Arikara ceramics
University of Illinois Press Urbana 1965

4 R G CHENHALL (editor)
Newsletter of computer archaeology
Department of Anthropology Arizona State University Tempe 1965 and later
Other useful sources include:
D HYMES
The use of computers in anthropology
Mouton & Co The Hague 1965
B WARREN
Computers and research in archaeology
Dittoed 1965
D F GREEN
Computer bibliography
Machine listing available on request from the author Weber State College Ogden Utah 1967

5 W D LIPE
Personal communication
1967

6 R G CHENHALL
The description of archaeological data in computer language
American Antiquity 32 161-67 1967

7 R G CHENHALL
An investigation of taxonomic systems for the storage and retrieval of material-culture data on electronic computers
Department of Anthropology Arizona State University Tempe 1965

8 A O SHEPARD
Ceramics for the archaeologist
Carnegie Institution of Washington Washington DC 1957

9 C R McGIMSEY D F GREEN
IBM ceramic code outline
University of Arkansas Museum 1965

10 A D KRIEGER
New world lithic typology project: part II
American Antiquity 29 489-493 1964

11 E M WEYER
New world lithic typology project: part I
American Antiquity 29 487-489 1964

12 L R BINFORD
A proposed attribute list for the description and classification of projectile points
University of Michigan Anthropological Papers 19 193-221 1963

13 J D JENNINGS
Information on University of Utah computer study
Department of Anthropology University of Utah 1964

14 E M STERN
Using the IBM 7090 in the classification of ground stone tools
Michigan Archaeologist 12 229-234 1966

15 J CHRISTOPHE J DESHAYES
Index de l'outillage sur cartes perforées: outils de l'age du bronze, des Balkans á l'Indus

Centre National de la Recherche Scientifique Paris 1964

16 J BIRD
The use of computers in the analysis of textile data; specifically archaeological fabrics from Peru
The American Museum of Natural History New York 1967

17 R P ROARK
From monumental to proliferous in Nasca pottery
Ñawpa Pacha 3 1-92 Dept. of Anthropology University of California Berkeley 1965

18 J MULLER
Style and archaeology
Department of Anthropology Southern Illinois University Carbondale 1967

19 I SCOLLAR
Personal communication 1967

20 Important recent papers on archaeological sampling include Vescelius G *Archaeological sampling a problem of statistical inference*
in Dole and Carneiro Essays in the Science of Culture in Honor of Leslie A White Thomas Y Crowell New York 1960 pp 457-70
S ROOTENBERG
Archaeological field sampling
American Antiquity 30 181-188 1964
G L COWGILL
The selection of samples from large sherd collections
American Antiquity 29 467-474 1964
L R BINFORD
A consideration of archaeological research design
American Antiquity 29 425-441 1964
J N HILL
Random sampling a tool for discovery
Department of Anthropology UCLA 1967
clude Vescelius G

21 J A BROWN L G FREEMAN
A UNIVAC analysis of sherd frequencies from the Carter Ranch Pueblo Eastern Arizona
American Antiquity 30 162-167 1964
L G FREEMAN J A BROWN
Statistical analysis of Carter Ranch pottery
Fieldiana: Anthropology 55 126-154 Chicago Natural History Museum 1964

22 A McPHERRON
Programming the IBM 7090 for optimizing taxonomy in archaeology
Department of Anthropology University of Pittsburgh 1963

23 W A LONGACRE
Archaeology as anthropology: a case study
Science 144 1454-55 1964

24 J D JENNINGS *op cit*

25 L R BINFORD S R BINFORD
A preliminary analysis of functional variability in the Mousterian of Levallois facies
American Anthropologist 68 no 2 part 2 238-295 1966

26 G L COWGILL
Evaluación preliminar de la aplicación de métodos a máquinas computadoras a los datos del mapa de Teotihuacán
Department of Anthropology Brandeis University Waltham Mass 1966

27 J N HILL
A prehistoric community in eastern Arizona
Southwestern Journal of Anthropology 22 9-30 Albuquerque 1966

28 R A BENFER
A design for the study of archaeological characteristics by population genetical and psychological models
American Anthropologist in press

29 F R HODSON P H A SNEATH J E DORAN
Some experiments in the numerical analysis of archaeological data
Biometrike 53 311-24 1966
J E DORAN F R HODSON
A digital computer analysis of Palaeolithic flint assemblages
Nature 210 688-89 1966

30 V ELISEEFF
Possibilités du scalogramme dans l'étude des bronzes chinois archaiques
Mathematiques et Sciences Humaines 11 1-10 1965

31 W F DE LA VEGA
Classification des tombes d'une nécropole d'Italie du sud, sur calculateur
Proceedings of the International Symposium on Computational and Mathematical Methods in the Behavioral Sciences Rome 1966 International Computation Centre Rome in press

32 Much the same point is made in a review of Deetz's work by Spaulding A C *American Anthropologist* 68 1064-5 1966

33 R S KUZARA G R MEAD K A DIXON
Seriation of anthropological data: a computer program for matrix-ordering
American Anthropologist 68 1442-55 1966

34 M ASCHER R ASCHER
Chronological ordering by computer
American Anthropologist 65 1045-52 1963

35 L JOHNSON
Towards a statistical overview of the archa cultures of Central and Southwestern Texas
Texas Memorial Museum Austin Bulletin 12 1967

36 R N SHEPARD
The analysis of proximities: multidimensional scaling with an unknown distance function
Psychometrika 27 125-140 and 219-246 1962

37 F G LOUNSBURY
A formal account of the Crow- and Omaha-type kinship terminologies
In Goodenough W editor Explorations in Cultural American Anthropologist 68 no 2 part 2 356-94 York 1964

38 J R SACKETT
Quantitative analysis of Upper Palaeolithic stone tools
Department of Anthropology State University of New 1966

39 W D LIPE C F HUNTINGTON
The application of some centrographic techniques to the analysis of archaeological data
Anthropology McGraw-Hill Book Company Inc New York at Binghamton 1964

40 I SCOLLAR F KRÜCKEBERG
Computer treatment of magnetic measurements from archaeological sites
Archaeometry 9 61-71 1966

41 Principal investigator of this project is René Millon
Papers concerning the computer work include
G L COWGILL
Computers and prehistoric archaeology
In E Bowles editor Computers in Humanistic Research Prentice-Hall Inc Englewood Cliffs New Jersey 1967 chap 6 pp 47-56
G L COWGILL
Computer archaeology at Teotihuacan Mexico 1965
Statistical and computer approaches to sociocultural interpretation of an ancient city of Mexico 1966
Evaluación preliminar de la aplicación de mátodos a máquinas computadoras a los datos del mapa de Teotihuacán
Department of Anthropology Brandeis University Waltham Massachusetts 1966

42 IU V KNOROZOV
Machine decipherment of Maya script
Soviet Anthropology and Archaeology 1 43-50 1962/3

43 J J RENDÓN A SPESCHA
Nueva clasificación plastica de los glifos Mayas
Estudios de Cultura Maya 5 189-280 Mexico City 1965

44 One is by Elizabeth W Barber Department of Linguistics Yale University The other is by Richard Morgan and John Reich Classics Dept University of Manitoba cited in Computers and the Humanities 1 233 1967

45 D F GREEN personal communication 1967

46 J W DOW
Astronomical orientations at Tectihuacán a case study in astro-archaeology
American Antiquity 32,326-334 1967

Computer applications in political science

by KENNETH JANDA
Northwestern University
Evanston, Illinois

INTRODUCTION

To some, "political science" is a contradiction in terms. They regard politics as an art which defies systematic study and, hence, offers no basis for a "science." Others contend, as I do, that human behavior is subject to systematic study, explanation, and prediction—and this includes man's *political* behavior. While the attitude of the professional student of politics toward this issue may still reveal his attitude toward the computer as a useful or even "legitimate" tool in his research, the argument over the "behavioral approach" in political science is fast becoming irrelevant to computer applications in political research. Not only is the computer becoming a "conventional" research tool in patently humanistic studies like literature,[1] music,[2] and art,[3] but it is also winning favor as a useful aid to hard-nosed professional politicians—witness the conference held in Chicago last spring on data processing for Republican party workers.[4]

Exactly *how* have computers been used in political science? This paper will try to answer the question by reviewing actual computer applications in three methodological categories: data analysis, information processing, and simulation. Within each of these categories, the discussion will proceed from the more frequent to less frequent usage of computers in political research. In this review, relatively little attention will be given to the techniques themselves—most of which are assumed to be familiar to the audience toward which this paper is directed. Instead, attention will be focused on substantive applications by citing publications of political scientists who have used computers in their research. These citations will be illustrative of the applications rather than exhaustive of the work done on the topic.

Data analysis

By far the most common usage of computers in political science is to analyze quantitative data on individual actors in the political process (e.g., voters, legislators, judges, etc.), aggregates of citizens (e.g., nations, states, cities, etc.), and political institutions (e.g., courts, political parties, legislatures, interest groups, intergovernmental organizations, etc.). A general introduction to recording and analyzing political data in punchcard form is contained in my book, *Data Processing: Applications to Political Research.*[5]

In large part, computer analyses of these data involve nothing more than the application of conventional statistical routines incorporated in general library programs.[6] For purposes of discussion, these routines will be separated into "bivariate" and "multivariate" analyses. But for some types of political analysis, existing statistical programs are of little use, which gives rise to the development and application of "special purpose" programs for political research. This section will review, in turn, "bivariate," "multivariate," and "special purpose" computer analyses of quantitative political data.

Bivariate analysis: Here, the term "bivariate analysis" includes all measures of association between two variables, be they nominal-, ordinal-, interval-, or ratio-scale variables.[7] For much of the data that interest political scientists, the computer is instructed merely to cross-tabulate one variable against another—sometimes calculating appropriate parametric or nonparametric statistics to summarize the extent of the correlation or sometimes providing only percentages to facilitate interpretation of the relationship. Occasionally the computer holds one or more other variables "constant" when cross-tabulating two variables, but this analysis is still essentially bivariate rather than multivariate.

Without question, the type of political data employed most frequently in bivariate analysis with a computer is generated through sample survey research. Questions about attitudes towards politics, voting intentions, and

sociological characteristics have been employed in countless studies of voting behavior and political participation. Although the results of 1,000 or more interviews with a national sample of the electorate are invariably recorded on punchcards, it should be pointed out that these studies are often analyzed with unit record equipment (i.e., a counter-sorter) rather than computer. This is especially true of the surveys done by the national polling organizations, e.g., Gallup and Roper.

While the counter-sorter is useful for processing the few questions asked by a commercial polling organization, it is rapidly overshadowed by the power of the computer when the number of questions is large, as in surveys conducted by academic research organizations like Michigan's Survey Research Center. The SRC's landmark study of voting behavior in the 1952 and 1956 presidential elections, *The American Voter,*[8] utilized approximately 2,000 interviews taken before and after each election, producing eight cards of data per respondent in the 1952 election and nine cards per respondent in 1956. For this increasingly popular form of research, involving many variables for a relatively large number of cases, the computer's talents are used to generate desired cross-tabulations and associated statistics vital to the researcher.

More recently within political science, the computer has been used in bivariate analyses of data collected on *nations* instead of individuals. The data represent such variables as gross national product, legislative-executive structure, number killed in domestic conflict, nature of the party system, literacy rate, and so on.[9] Some of these variables are patently quantitative in nature, others involve qualitative categories.

Representing one approach to computer analysis of such data, Banks and Textor's *A Cross-Polity Survey*[10] expanded a total of 57 quantitative and qualitative variables to 177 different dichotomizations of the variables across 115 countries or "polities." They instructed the computer to cross-tabulate every dichotomized variable against every other dichotomized variable, printing out only those fourfold contingency tables that were statistically significant at the .10 level. The 1,200 page *Cross-Polity Survey,* reproducing the computer output from this analysis, was published as a reference source for political scientists seeking the relationship between basic variables on countries across the world. In the case of such cross-national research, the number of units under study does not justify the use of the computer as much as its power and flexibility in cross-tabulating variables for analysis.

Multivariate analysis: Although bivariate analysis constitutes the most common usage of computers in political research, the above section does not dwell on those applications because bivariate analysis is assumed to be relatively routine and uninteresting to those outside of political science who are already familiar with computers. A far more interesting application of computer technology lies in *multivariate* analysis of quantitative political data.

For some reason, factor analysis has been the most popular multivariate technique reported in the recent political science literature within the last seven years. One condition which accounts for the popularity of factor analysis in political research is the ready availability of suitable computer programs. As Pinner noted in 1960:

> *For decades, factor analysis has been the exclusive domain of experts; the mathematical sophistication needed and the inordinate amount of labor often required made it prohibitive to the ordinary researcher in social science. Recently these difficulties have been largely removed. We have now good introductory descriptions of the method written for people of moderate statistical means. Moreover, the presence of high-speed computers on most larger campuses and the existence of "canned" programs has taken most of the work out of factor analysis.*[11]

Another condition for political scientists' focus upon factor analysis as *the* multivariate technique most often applied to political data is the relatively low state of theoretical development within most fields of research. Factor analysis, which discloses the relationships that underlie an intercorrelation matrix for large numbers of variables, is well suited to "fishing expeditions" when the researcher has few hypotheses to guide his search for relationships among variables.

A lengthy review of substantive findings from various factor analyses of political data is available elsewhere,[12] and I will confine my treatment to illustrating the range of applications within political science. Factor analysis has been applied to roll call voting in the United Nations General Assembly,[13] the United States Congress,[14] various state legislatures,[15] and the French Chamber of Deputies;[16] decisions and opinions in the Supreme Court;[17] survey interview responses;[18] interaction patterns observed within local government bodies;[19] domestic and foreign conflict behavior within and between nations;[20] election returns and demographic variables by geographical areas;[21] and even attitudes of political scientists toward their profession.[22] In fact, the use of factor analysis in political research has been sufficient to cause one political scientist, Rudolf J. Rummel, to write a textbook on the subject.[23]

For right or wrong, factor analysis has emerged as the principal multivariate technique in political science—much as the analysis of variance has emerged as the

major statistical technique in psychology. It is interesting to note that the conditions which promote usage of analysis of variance in psychology—specific hypotheses to be tested and controlled experimental conditions—are largely absent in political research, making for a noticeable dearth of research in political science based on multiple-way analysis of variance.

Following some distance behind factor analysis, the most frequent type of multivariate computer program used in political research is multiple regression analysis —usually reported only as multiple correlation. More recently within political science, attention has been given to other techniques of multivariate analysis now available through standard library computing programs. In particular, canonical correlation and discriminant analysis have been applied to political data.[24] Moreover, a great surge of interest has developed in the causal inference techniques that Simon[25] and Blalock[26] have introduced into social research. While programs to construct and evaluate alternative causal "models" of political phenomena are not yet standard library items at most computing centers, they soon will be, and the amount of work done with causal inference can be expected to mushroom as these programs become available.

Special purpose analysis: The causal inference programs mentioned above are not what I regard as special purpose programs for political analysis. Causal inference is a general technique applicable to many types of data and indeed came into political science from sociology.[27] On the other hand, some problems of political research are defined by the nature of the data and have relatively few counterparts outside of political science. One clear instance of this is roll call analysis; another example somewhat less exclusively within the province of political scientists is the analysis of transaction flows.

The recent book by Anderson, Watts, and Wilcox, *Legislative Roll-Call Analysis,*[28] establishes the place of the methodology within political science and contains a set of four computer programs specifically designed for the analysis of roll call votes. The tasks these programs perform with roll call data, apart from the factor analysis application mentioned above, bear little relation to standard statistical techniques. For instance, one program reads the voting positions of all the legislators on a given set of issues in the Congress or state legislatures and produces the Democrat-Republican division on each issue, an index of cohesion within each party, an index of party likeness on the bill, and Riker's coefficient of significance for the vote. Another program gives the Riker coefficient of significance from marginal divisions when individual votes are not available; a third generates fourfold cross-tabulations with corresponding correlation coefficients, measures of cohesion, and measures of party likeness; and the last program calculates the Lijphart Index of Agreement for pairs of nations casting votes in the United Nations General Assembly.

Although transactions between units of analysis (e.g., notes transmitted between office workers, smiles directed toward friends of the opposite sex, goods traded between Indians) may be of interest to other social scientists, the political scientist has a special stake in analyzing transaction flows. Large amounts of public data exist on the political, economic, social, and cultural relationships among governmental units—especially nations in the international system. The availability of this information and interest in international relations has stimulated political research into transaction flows. Findings from transaction flow analyses among nations in the form of diplomatic exchanges, trade, and shared membership in intergovernmental organizations. Brams wrote his own computer program to analyze his transaction data,[30] and also used another computer program for the hierarchical decomposition of his transaction flow matrices to identify subgroups of nations most closely linked together.

Information processing

In contrast to "data analysis," where input to the computer is in numerical form, "information processing" utilizes the computer capabilities for accepting natural language text as input. This is an important feature of the computer for political scientists, whose material often resists easy quantification. Consequently, we find a considerable amount of research being done with the computer on content analysis of political documents. In addition to analyzing the content of natural language text, the computer is also being used with increasing frequency to search and retrieve information from textual material in machine readable form.

Content analysis: There is something about political documents—be they speeches, tracts, treaties, or diplomatic messages—that invites content analysis.[31] Many non-political scientists who write computer programs for content analysis elect to apply their programs to political texts. Thus we find non-political scientists using the computer to determine the authorship of disputed papers in *The Federalist,*[32] to locate ambiguities in the Nuclear Test Ban Treaty,[33] and to score speeches by Castro, Kennedy, and Nixon on a set of themes or concepts.[34]

Notwithstanding these varied approaches to the subject, by far the most concerted work in content analysis of political documents has been done with the use of a computer program called "The General Inquirer," which was developed by Philip J. Stone and his associates.[35] As a social psychologist, Stone was not explicitly concerned with the application of his program to political documents, although he and his associates did carry out some of this work. Within political science, Holsti has concen-

trated the most on research applications of the General Inquirer.[36]

The General Inquirer requires some editing of the text before analysis. This amounts to chopping off "ed" and "ing" endings and, in some applications, adding subscripts to important words in the sentence to identify the "perceiver," "agent," "action," "target," and so on. The General Inquirer then matches each work in the text against a dictionary of terms for scoring purposes. Holsti created a dictionary of 4,000 political terms, each of which was rated along dimensions of affect, strength, and activity. For example, the word "abandon" might be rated *negative, weak,* and *passive.* The program has several capabilities for content analysis, including the generation of statistics concerning the appearance of words in the text as rated along the scales built into the dictionary; the identification of themes according to use of certain words or dimensions; and the indexing of certain words in the text. Holsti has applied this computer system for content analysis to messages between key decision makers in the 1914 crisis and to communications during the Cuban missile crisis of October 1962.[37]

Information retrieval: Computer programs for content analysis are designed to evaluate messages contained in a relatively small amount of material. Information retrieval programs, on the other hand, have the function of searching relatively large amount of material and delivering on command specified subsets of that material. An introduction to some basic techniques in information retrieval is contained in my *Information Retrieval: Applications in Political Science.*[38] Within political science, information retrieval programs have been applied to bibliographical material, propositional inventories, and descriptions of studies and variables stored in local "data libraries." Examples of each application will be presented.

Computer programs for indexing bibliographical material according to "keywords" in titles or annotations have been applied to the behavioral sciences[39] with results comparable to their success in the physical sciences.[40] Specifically within political science, keyword-in-context (KWIC) indexing has been used to compile a cumulative index to all articles published in *The American Political Science Review* from 1906 through 1963.[41] Another index has been published for all articles in the *Midwest Journal of Political Science* from 1957 through 1967.[42] A far more ambitious approach to indexing the literature on political science is exemplified by the Universal Reference System, whose scheduled "Government and Public Policy Series" is designed as a ten-volume comprehensive index to virtually all fields of political science.[43] Volume I of this series, *International Affairs,* has already been published.[44]

Another approach to computer retrieval of bibliographic material involves searching abstracts of documents according to logical combinations of keywords. This approach is being followed at Northwestern University, where a computer program called TRIAL (for *T*echnique to *R*etrieve *I*nformation from *A*bstracts of *L*iterature)[45] is being used in a "selective dissemination of information" system for political scientists and other faculty members interested in cross-national and interdisciplinary studies.[46] A similar system is in operation at the University of Georgia, where political science students and faculty cooperate in contributing abstracts to a file that can be searched with the TRIAL program.[47]

It is obvious that computer programs for keyword indexing and the retrieval of logical combinations of keywords need not be limited to bibliographic material but may be applied to different types of textual input. Within political science, these techniques have been employed in building and managing inventories of propositions dealing with political participation[48] and political parties.[49] In these applications, researchers identify and formulate the propositions, which are then keypunched for computer processing. Computer-generated keyword indexes to variables in the propositions can be helpful in building thesauri and clarifying terms. Programs for searching the inventory and retrieving desired propositions provide a method for *using* the propositions once collected.

One of the most promising applications of information retrieval techniques within political science—or within the social sciences generally—is in providing access to data gathered by other researchers and deposited in data repositories or "libraries." The research interests of political scientists call for a bewildering variety of data gathered across time for political institutions and classes of people across the world. The vast amount of time required to collect, check, and keypunch almost any set of political data—e.g., roll call votes in a state legislature—emphasizes the need for constructing user-oriented libraries for storing and disseminating machine readable data.

This need has been felt most clearly by those engaged in sample survey research who want to make more effective use of the data gathered in literally thousands of polls and surveys taken in the U.S. and abroad. Most effort toward developing systems to retrieve political data has been focused on locating surveys containing questions of interest to the researcher. The goal of such systems is not only to identify the survey, but to furnish the researcher with the actual questions and the column locations in the data cards indicating where the responses are recorded. One such system for retrieving interview questions from sample surveys employs the General Inquirer;[50] another adapts the TRIAL program for this purpose.[51]

Of course, the idea of retrieving data for specific research needs can be generalized beyond sample survey data. Although the accomplishments here are not as impressive, one can cite the use of keyword indexing for locating the substantive issues on which roll call votes were taken in state legislatures, the U.S. Congress, and the United Nations General Assembly.[52] Working with mainly quantitative data, Beck and Stewart have developed their own routines for retrieving biographical information on Eastern European elites.[53] An effort is under way at Northwestern University to develop a system for retrieving desired studies and variables from a variety of studies, including sample surveys.

Simulation

From the standpoint of substantive political science, the computer's major contribution may well come from what is so far its least common application: simulation of political processes. While computer simulation is fast becoming a conventional tool of industrial engineering and has entrenched itself within the social sciences in economics and psychology, it is just emerging as an aid to theory construction in political science.

Interestingly, one of the earliest successes in computer simulation of social phenomena occurred in political science. Late in the summer of 1960, Ithiel de Sola Pool and his associates simulated the results of the forthcoming presidential election if Kennedy were to meet the religious issue head on in his campaign.[54] Using data collected from some fifty national sample surveys between 1952 and 1958 (an interesting use of a "data library" in itself), Pool and associates constructed 480 voter "types" based on seven different variables. They then instructed the computer to apply one of several sets of calculations for each of these voter types, under the assumption that party identifiers and religious groups would be affected in certain ways because of the religious issue. The results of their simulation correlated .82 with the actual vote for Kennedy, when the election was held months later.

The value of simulation as a theory building tool was not demonstrated until later, when essentially the same model was applied to the 1964 election of Johnson versus Goldwater. In the course of adjusting the equations and parameters for the new electoral situation, vague notions about the campaign and expected voting behavior became articulated, and the causal mechanism of the election became clearer. Because the computer will not tolerate vagueness, the researcher must understand his theory and the extent of his knowledge to write a simulation program. By enabling the researcher to test alternative theories, the computer offers him a way to improve his thinking and his theory.

An even better example of the use of the computer to build theory about the political process may be found in Shapiro's and Cherryholmes' computer simulation of the U.S. House of Representatives.[55] Shapiro and Cherryholmes devised a simulation that had both deterministic and stochastic elements. In the deterministic phase, each member of the House in the 88th Congress was confronted with selected bills dealing with foreign affairs and social welfare. As his background characteristics were matched against coded information on the bill, the representative was scored as "predisposed" to vote for or against each bill. Those men who were strongly predisposed for or against were assumed to vote that way; those who were not strongly predisposed in either direction then passed into the stochastic "communication" phase of the simulation.

In this phase, each "undecided" legislator "talked" with a certain number of other legislators, thus exposing himself to their predispositions and thus their "influence." *Who* the representative talked to was determined by calling random numbers matched against probabilities attached to the relationship of the representative with every other legislator. For example, the probability was higher that he would talk to someone of his own party, from his geographical region, on his committee, etc. These values or parameters entered into the program were drawn from available research studies on interactions within legislative bodies. Up to now, these studies offered little more than isolated findings, but Shapiro and Cherryholmes were able to incorporate their findings into a theory of the communication process within Congress. Even in this initial attempt, their success in building theory about the political process can be judged from the fact that they correctly predicted the vote of the legislators in 85% of the cases and that their estimates of the House's for/against split on each of some 50 bills correlated above .95 with the actual votes.

SUMMARY

Within political science, computers have been most frequently used for relatively standard statistical analysis of quantitative data. While the statistical treatment usually encompasses only bivariate analysis, it sometimes extends to multivariate techniques — especially factor analysis. The analysis of political data also requires some special purpose programs, particularly in the case of roll call data, but in this sense political science may not be any more demanding than the other social sciences and probably less demanding than the physical sciences.

Apart from statistical analysis, computers have had numerous applications within political science to problems of information processing. Heading this list of applications is content analysis, much of which has been done using the General Inquirer program. Keyword indexing of titles, selective dissemination of information,

and retrospective searching of abstracts have also been done with bibliographical material in political science. One of the most promising applications of information retrieval techniques lies in the improved utilization of data libraries formed to house collections of studies.

Finally, computer simulation of political processes, while not yet a standard research technique within political science, is certain to be used more in the future—judging from the impressive results of some simulations that have been done.

REFERENCES

1 L T MILIC
Computer and the humanities
149-150 March 1967

2 G BERLIND B S BROOK
Ibid 150-152

3 L MEZEI
Ibid 154-156

4 *Electronic data processing training conference*
Sponsored by the Republican National Committee Chicago 20-21 April 1967

5 Evanston: Northwestern University Press 1965

6 W J DIXON ED
BMD: Biomedical computer programs
Los Angeles School of Medicine University of California 1964

7 This distinction among different types of measurement or scaling is elaborated in Janda, *op cit* 168-171

8 A CAMPBELL P E CONVERSE W E MILLER D E STOKES
New York John Wiley 1960

9 B M RUSSETT ET AL
World handbook of political and social indicators
New Haven Yale University Press 1964
R J RUMMEL
The dimensionality of nations project
In R MERRITT S ROKKAN EDS
Comparing nations: the use of quantitative data in cross-national research
New Haven Yale University Press pp 109-129 1966

10 A S BANKS R B TEXTOR
Cambridge M.I.T. Press 1963

11 F A PINNER
Notes on method in social and political research
In D WALDO ED
The research function of university bureaus and institutes for government-related research
Bureau of Public Administration University of California Berkley p 205 1960

12 R L MERRITT
Political science and computer research
In E A BOWLES ED
Computers and humanistic research
Englewood Cliffs NJ Prentice-Hall pp 90-107 1967

13 H R ALKER JR.
Dimensions of conflict in the general assembly
American Political Science Review 58 642-657
September 1964

14 C D McMURRAY
A factor method for roll call vote studies
American Behavioral Scientist 6 26-27 April 1963

15 J G GRUMM
A factor analysis of legislative behavior
Midwest Journal of Political Science 7 336-356
November 1963

16 D MACRAE JR
Intraparty divisions and cabinet coalitions in the Fourth French Republic
Comparative Studies in Society and History 5 64-211
1963

17 G SCHUBERT
The judicial mind
Evanston: Northwestern University Press 1965

18 C D McMURRAY M B PARSONS
Public Attitudes Toward the Representational Roles of Public attitudes toward the representational roles of legislators and judges
Midwest Journal of Poliitcal Science 9 167-185
May 1965

19 J D BARBER
Power in committees: an experiment in the governmental process
Chicago Rand McNally 1966

20 R J RUMMEL
Dimensions of conflict behavior within and between nations
General Systems Yearbook 8 1-50 1963

21 E ALLARDT
Social sources of Finnish communism: traditional and emerging radicalism
International Journal of Comparative Sociology 5 49-72
1964

22 A SOMIT J TANNENHAUS
American political science: a profile of a discipline
New York Atherton 1964

23 *Applied factor analysis*
Evanston Northwestern University Press (forthcoming)

24 B RUTHERFORD
Canonical correlation and political analysis
Evanston Department of Political Science Northwestern University (unpublished paper) 1966
I ADELMAN C T MORRIS
'Self-Help' criteria for underdeveloped countries: an operational approach
Evanston Department of Economics Northwestern University (unpublished paper) 1967

25 H A SIMON
Models of man
New York John Wiley especially chaps 1 to 3 1957

26 H M BLALOCK JR
Causal inferences in nonexperimental research
Chapel Hill University of North Carolina Press 1961

27 H M BLALOCK JR
Causal inferences, closed populations, and measures of association
American Political Science Review 61 130-136
March 1967

28 L F ANDERSON M W WATTS JR A R WILCOX
Legislative roll-call analysis
Evanston Northwestern University Press 1966

29 *Op cit* 23-26
30 S J BRAMS
Transaction flows in the international system
American Political Science Review 60 88-898 December 1966
31 R NORTH ET AL
Content analysis
Evanston Northwestern University Press 1963
32 F MOSTELLER D L WALLACE
Inference and disputed authorship: The Federalist
Reading Mass Addison-Wesley 1964
33 R A LANGEVIN M F OWENS
Computer analysis of the nuclear test ban treaty
Science 146 11-86-1189 27 November 1964
34 A R CARLSON
Concept frequency in political text: an application of a total indexing method of automated content analysis
Behavioral Science 12 January 1967 68-72
35 P J STONE D C DUNPHY M SMITH
D M OGILVIE
The general inquirer: a computer approach to content analysis—studies in psychology, sociology, anthropology and political science
Cambridge MIT Press 1966
36 O R HOLSTI
An adaptation of the 'general inquirer' for the systematic analysis of political documents
Behavioral Science 9 332-338 October 1964
37 O R HOLSTI R A BRODY R C NORTH
Violence and hostility: the path to world war
Stanford University Studies in International Conflict and Integration February 1964 (multilith)
Theory and measurement of interstate behavior: a research application of automated content analysis
Stanford University Studies in International Conflict and Integration May 1964 (multilith)
O R HOLSTI
Computer content analysis in international relations research
in E A BOWLES ED
Computers in Humanistic Research
Englewood Cliffs NJ Prentice-Hall pp 108-118 1967
38 Indianapolis Bobbs-Merrill (in press)
39 K JANDA
Keyword indexes for the behavioral sciences
American Behavioral Scientist 7 55-58 June 1964
40 M FISCHER
The KWIC index concept: a retrospective view
American Documentation 17 57-70 April 1966
41 K JANDA ED
Cumulative index to the American political science review volumes 1-57: 1906-1963
Evanston Northwestern University Press 1964
42 K JANDA NEAL E CUTLER
Cumulative index to the Midwest journal of political science, volumes 1-10: 1957-1966
Midwest Journal of political science 11 225-255 May 1967
43 A DE GRAZIA
Continuity and innovation in social science reference retrieval: illustrations from the universal reference system
American Behavioral Scientist 10 1-4 February 1967
44 20 Nassau St Princeton N J 08540
Universal Reference System 1965
45 K JANDA W H TETZLAFF
TRIAL: a computer technique to retrieve information from abstracts of literature
Behavioral Science 11 480-486 November 1966
46 K JANDA G RADER
Selective dissemination of information: a progress report from Northwestern University
American Behavioral Scientist 10 24-29 January 1967
47 W A WELSH
The TRIAL system: information retrieval in political science
American Behavioral Scientist 10 11-24 January 1967
48 L W MILBRATH K JANDA
Computer applications to abstraction, storage, and recovery of propositions from political science literature
Paper Annual Meeting of the American Political Science Association Chicago 1964
49 K JANDA
Retrieving information for a comparative study of political parties
In W J CROTTY ED
Approaches to the study of party organization
Boston Allyn and Bacon (in press)
50 E K SCHEUCH P J STONE
The general inquirer approach to an international retrieval system for survey archives
American Behavioral Scientist 7 23-28 June 1964
51 A R WILCOX D B BOBROW D P BWY
System SESAR: automating an intermediate stage of survey research
American Behavioral Scientist 10 8-11 January 1967
52 K JANDA
Information retrieval: applications to political science
Op cit Chap 8
53 C BECK D K STEWART
Machine retrieval of biographical data
American Behavioral Scientist 10 30-32 February 1967
54 I DE SOLA POOL R P ABELSON S POPKIN
Candidates, issues, and strategies: a computer simulation of the 1960 *and* 1964 *presidential elections*
Cambridge M.I.T. Press 1965
55 C H CHERRYHOLMES
The House of Representatives and foreign affairs: a computer simulation of roll-call voting
Northwestern University 1966 (unpublished doctoral dissertation)
M J SHAPIRO
The House and the federal role: a computer simulation of roll-call voting
Northwestern University 1966 (unpublished doctoral dissertation)

The B8500-microsecond thin-film memory

by RICHARD H. JONES and ERIC E. BITTMANN
Burroughs Corporation
Paoli, Pennsylvania

INTRODUCTION

The computer in a B8500 modular data processing system may be equipped with as many as 16 memory modules, each of which has a capacity of 16,000 words of 52 bits each. The modules are self-contained with power supplies, logic circuits, and receiving and transmitting circuits which communicate with the computers.

The storage unit in a memory module is mounted in a single cabinet housing four memory frames with planar, ferromagnetic film storage elements. The films are interrogated in a linear select mode, in which a destructive "read" is followed by a "write" or a "restore" cycle. The half-microsecond memory cycle was chosen for reasons of economy and is not the upper frequency limit of a film memory.[1,2,3] In addition, four 52-bit words are always read during one memory cycle (fourfetch), because one thin-film word line embraces 208 bits.

The memory cell

The memory cell configuration was chosen which combined the film memory's characteristic high-speed switching behavior with a low-cost assembly technique. The packing density chosen yielded a reasonable sense signal and still allowed the employment of mass soldering techniques for the interconnection between the stack and electronic circuits.

The basic cell is assembled from paired substrates in which film spots are placed face-to-face, but separated by the word, digit, and sense conductors in a triplate arrangement. The separation between films is about 4 to 5 mils and tests have shown that magnetic coupling reduces the shape anisotropy by about 50 percent. The substrate pairs are enclosed with continuous ground sheets which provide the return conductors for the triplate arrangement of the word, digit, and sense line conductors. The triplate has the advantage that the fields generated by the return currents in the grounds cancel in the vicinity of the film cells, thereby reducing the effects of the disturbing field emanating from the neighboring word and digit lines. A disadvantage of the triplate is that because of this field cancellation effect, higher currents than those needed in the conventional strip line are required to generate a given field in the center conductor. However, improved memory cell performance and ease of assembly more than justify the increased current demand.

The magnetic thin-films are produced by vacuum desposition of Molypermalloy onto 3-mil thick glass substrates. The film spots are obtained by chemical etching. The spots are placed on 25-mil centers in the word direction and on 50-mil centers in the digit direction. A single glass substrate contains 3072 film rectangles.

Memory plane

Memory cross section

The memory cell previously described is mechanized as shown in Figure 1. The sense-digit lines and the word lines are fabricated from 1-mil-thick Kapton,* 1/2-ounce copper laminate. The sense-digit configuration and the word-line configuration are etched on separate pieces of laminate. The magnetic films on 3-mil-thick glass substrates are bonded to both sides of the sense-digit and word line laminates. Properly spaced copper ground planes complete the package.

Lattice construction

The basic element of the memory plane is the lattice assembly, as shown in Figure 2. Five sense-digit line tapes and four word line tapes are aligned and taped to a master with the ends of each tape precisely positioned. The group of tapes is then laminated into a lattice in a laminating press to assure uniformity of the cross section of the assembly. Next, 3-mil glass substrates are precisely positioned and laminated to

*A registered trademark of the E. I. du Pont Company

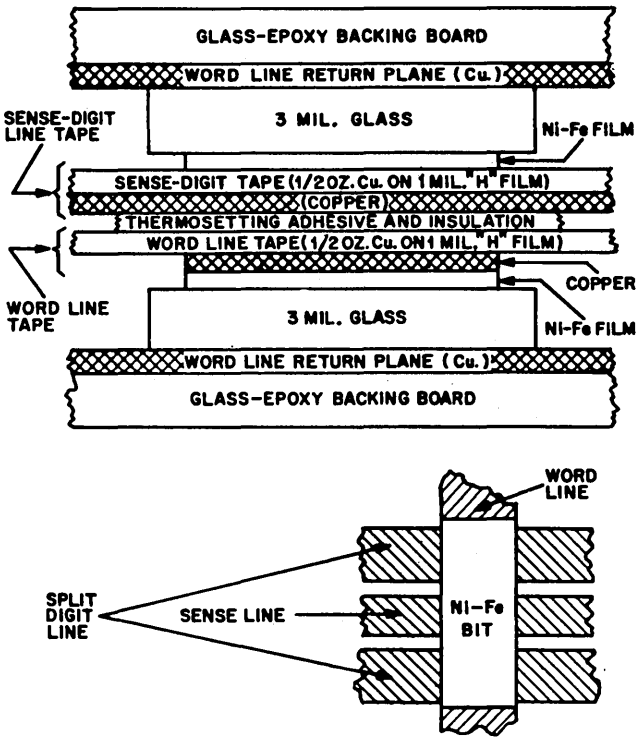


Figure 1 – Memory cell crossection

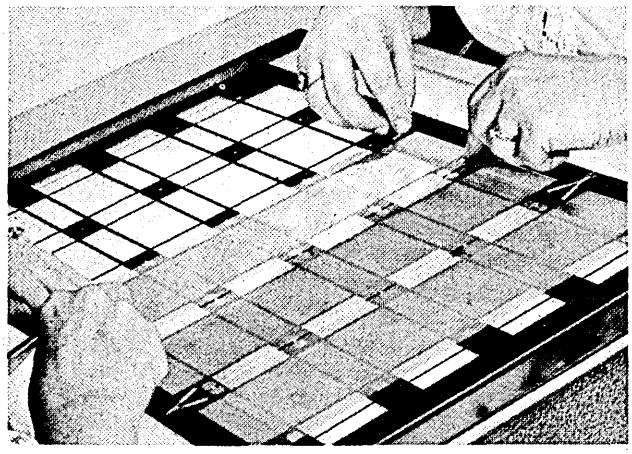


Figure 2 – Assembly of lattice

the lattice. Assembly criteria require that all portions of each intersection of a sense-digit group and a word line be covered by the corresponding magnetic bit. Special care is required to keep the magnetic film in intimate and consistent contact with the lattice. A gradated system of adhesives is used to maintain alignment during later laminating operations: (1) the copper-to-Kapton bond requires the highest temperatures, (2) the word-line-to-sense-digit tape adhesive system is intermediate, and (3) the substrate-to-lattice laminating system requires the lowest temperature.

The sense-digit cross section

Figure 3 illustrates the basic relationship of the sense and digit lines. This configuration was chosen to allow packaging of the digit drivers and sense amplifiers at opposite ends of the plane. The sense line crossover provides digit write noise cancellation. Assembly of four of the previously described lattices is illustrated in the exploded view. The triplate transmission line is mechanized with two outside ground planes (2-ounce copper laminated to glass-epoxy backing boards) and a solid copper inside ground plane. The inside and outside ground planes are electrically tied together at the periphery of the lattice.

The sense-crossover and digit feedthrough is made by using a three-level multilayer board assembly. The end-around functions are made by using an etched section of Kapton laminate wrapped around a glass-epoxy backing board. Sense-digit lines are terminated in printed-circuit boards which provide the connector interface to the digit drive and sense circuitry.

Word line cross-section

Figure 4 illustrates the basic relationship of the word lines in the memory plane. Triplate transmission line characteristics are maintained with the same inner and outer ground planes as previously shown in the sense-digit cross-section. The shorted end-word line is fabricated by soldering the bussed end of each word line tape to the inside ground. The word lines are placed on 25-mil centerline connectors but, because of the relative unreliability of 25-mil centerline connectors, the word lines are permanently tied to a printed circuit board which holds an 8×8 word-selection transistor matrix that permits a reliable connector interface on 0.100-mil centerlines.

Plane assembly techniques

Although the connector interface of the memory frame utilizes well-proven commercial connectors, the great number of solder connections necessary within the plane required a new connection technique that was both reliable and economical. The design of the memory plane internal connections was standardized using a process that can simultaneously solder many joints at a time. In each case, the end of a copper-Kapton laminated tape is reflow-soldered to a mating printed-circuit board. The etched copper-

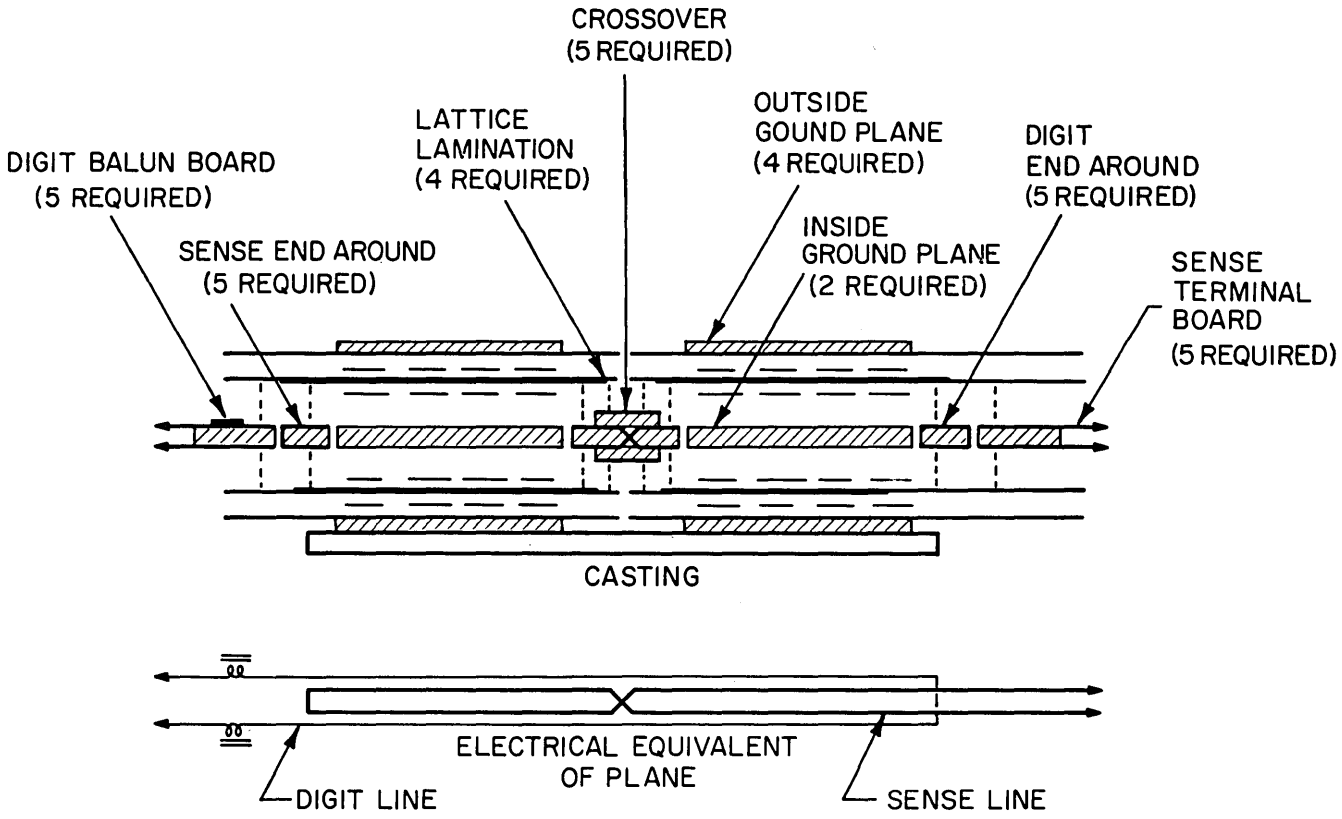


Figure 3 – Sense-digit lines crossection

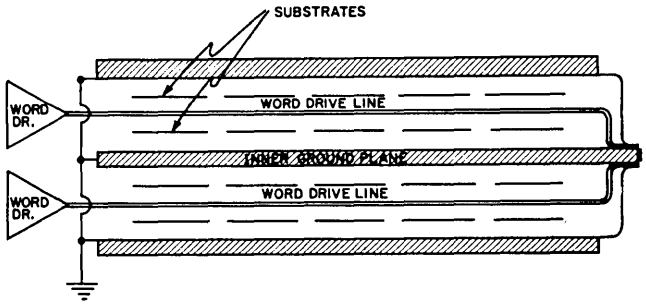


Figure 4 – Word line crossection

Kapton laminate and the printed-circuit board are then solder-plated to a closely controlled thickness and the plating is reflowed in hot palm oil.

A soldering machine was designed to apply a soldering blade to the Kapton side of the copper-Kapton laminate with the proper heat, timing, and distribution of pressure. The machine also provides the back-up and hold-down devices to ensure consistent, reliable reflow-solder joints. The heat is transferred through the Kapton to the plated conductors on the other side. These conductors have been previously aligned with plated conductors on the printed-circuit word. The temperature is adjusted for reflow of the solder at the interface of the conductors to be joined. To provide the maximum strength and reliability of the joint and a minimum of solder splashing between conductors, the following must be carefully controlled: temperature, pressure, cooling cycle, method of hold-down while cooling, and quality and thickness of solder plating. The plane can be indexed under the soldering head to previously set stops. The solder blade applications required for intraconnection of the plane can be made in less than one hour.

Memory circuits

The electrical interface and the logic functions are made using CTμL microcircuits. The current drivers and sense amplifiers are assembled in hybrid form utilizing Cermet silk-screened register-conductor patterns on alumina substrates. Semiconductors and capacitors are hand-soldered to the circuit chips.

Word drivers

Current to each word line is supplied by 1024 transistor switches which form a 32 × 32 matrix. The matrix is packaged on 16 multilayer cards containing 64 transistors each.

The 32 emitter drivers and the 32 base drivers energize the appropriate rows or columns in this matrix. All transistor switches are normally reverse-biased. Word current flows in a selected line when the emitter-output transistor saturates and supplies a negative current pulse to the appropriate emitter lines while the associated base driver removes the reverse bias from the selected transistor switch. A resistor between the emitter-driver output and the matrix controls the current amplitude. The word current has an amplitude of 600 ma ±10 percent and a rise and fall time of 15 to 20 ns with a duration of 150 ns.

The 32 emitter drivers and the 32 base drivers are packaged 8 to a card onto 4 printed circuits each. These 8 driver cards and the 16 word-matrix cards are interconnected through a multilayer backplane which forms part of the memory frame. (Refer to the subsequent paragraph and illustration describing the memory frame.)

Digit drivers

The digit drivers provide currents of either polarity and determine the future state of each memory cell after the word has been written into the stack. Digit current turn-on occurs while the word current flows and ends after the word current terminates.

The digit currents are supplied from two saturated-output transistors: one a PNP type for positive currents and one a NPN type for the negative currents.

The digit lines are shorted and resistors placed between the output transistor and the digit line control the current amplitude. The digit current has an amplitude of 150 ma ±10 percent, and a rise and fall time of 20 ns with a duration of 100 ns.

Sense amplifier

The nominal sense signal of ⅓ to ½ millivolt amplitude is amplified first by two differential stages followed by a threshold amplifier. The differential stages and the threshold amplifier are packaged on one hybrid chip each. The differential amplifier has over 45 db common rejection, eliminating the need for transformers at the amplifier input.

Memory frame

The memory frame shown in Figure 5 is made up of the memory plane and the memory circuits. Be-

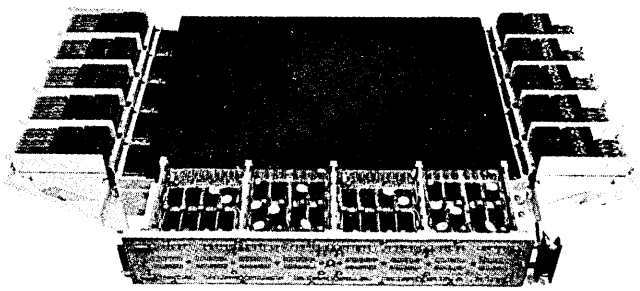


Figure 5 – Memory frame

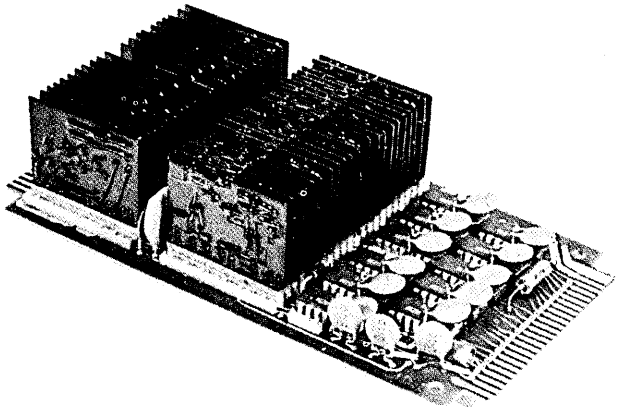


Figure 6 – Sense amplifier board

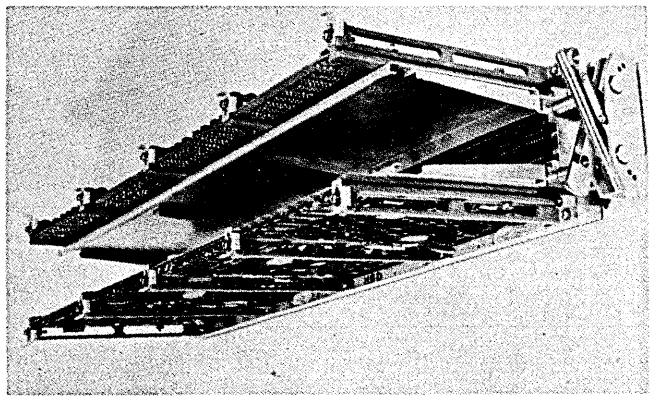


Figure 7 – Word address assembly

cause of the large ratio of power impressed on the memory to power-out of the memory and the proximity of conducting lines that allow crosstalk to occur. the memory electronics must be placed physically as close to the plane as possible. For this reason, the digit drivers, the sense amplifiers, and the word-address drivers are packaged integrally with the plane to form a memory frame. Each 1024-word X 208-bit frame has an interconnection level at the logic level (Fairchild CTμL).

The sense and digit circuit packaging is illustrated by the typical assembly shown in Figure 6. The sense amplifiers and digit drivers are each packaged on a hybrid circuit. The hybrid assemblies are plugged into receptacles on the mother board. The sense amplifiers and digit drivers are packaged on mother boards in groups of 22.

The 32 × 32 matrix which forms the 1024-address system for the memory is packaged as shown in Figure 7. The assembly interconnects the 8 × 8 matrix selection boards, which are a part of the plane, into a 32 × 32 matrix. The assembly also houses the 32 base and 32 emitter drivers which drive the word selection matrix. The interconnections for the selection matrix are fabricated using a 12-level multilayer printed-circuit board which maintains all interconnecting lines at 100 ohms characteristic impedance.

Memory organization

The 16,000 words of 52 bits each are stored in 4 thin-film memory planes as shown in the block diagram, Figure 8. Each plane has a capacity of 1024 words of 240 bits each, of which 208 bits are activated and the remaining 32 bits act as spares if needed. A film word line stores four computer words and every read cycle interrogates four words.

A 1024-word film plane contains word drivers, selection matrix, digit drivers, and sense amplifiers. The four planes share the address register, the information register, and the timing and controls circuits.

New words are written into the memory in 52-bit groups. If desired, all 208 bits can be loaded into the information register in four steps requiring 300 ns, after which all 208 bits are written into the selected address.

Memory timing

A memory cycle (Figure 9) begins with an initiate pulse and the gate pulse is sent to the selected base drivers 75 ns later. The emitter drivers receive their selected gate pulse at 100 ns. Emitter current to the matrix flows at 130 ns. Word current in the selected line interrogates the films at 140 ns. Sense signals appear at the differential amplifier output stages at 160 ns. The strobe pulse gates the sense signals into

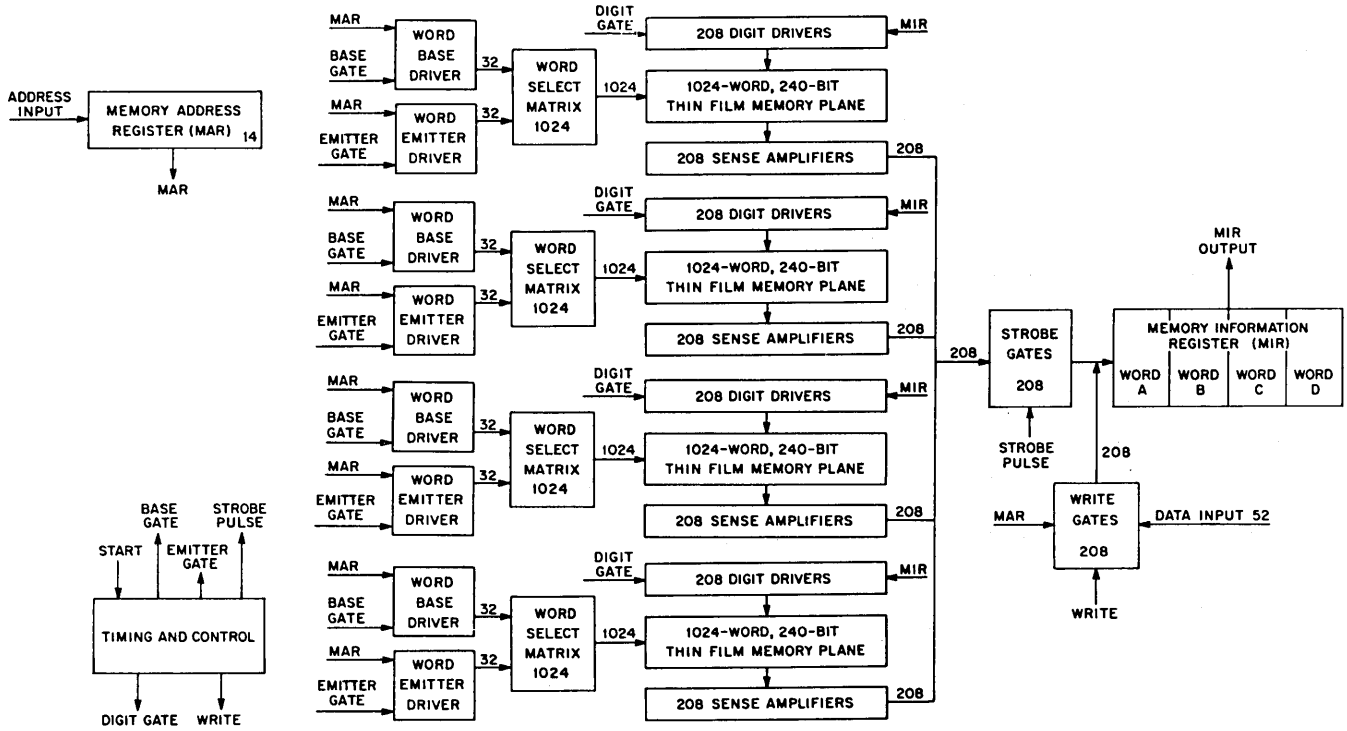


Figure 8 – Thin-film memory block diagram

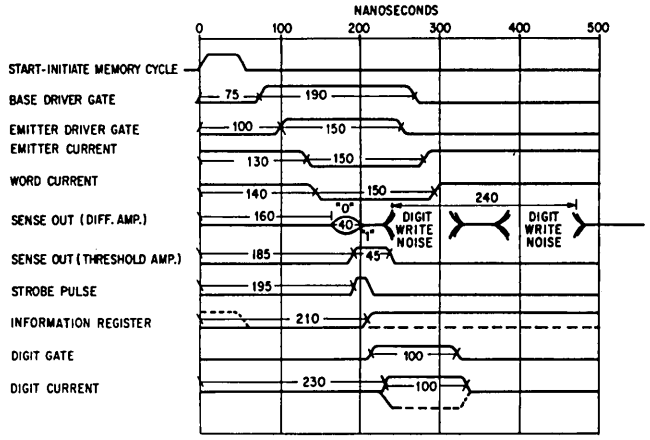


Figure 9—Thin-film memory timing diagram

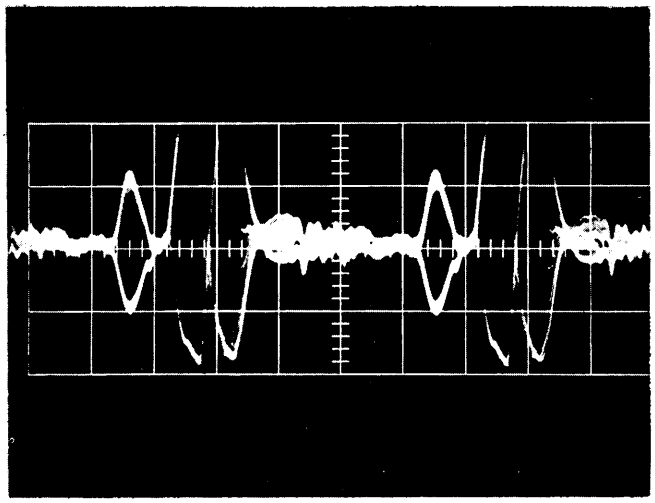


Figure 10—Sense signal waveform

the memory information register at 195 ns. The read data appears at the information register output before 210 ns. The restore or write cycle begins with the digit gate pulse at 215 ns. Digit current begins to flow in the plane at 230 ns and lasts for 100 ns. Word current turn-off occurs before digit current termination at 285 ns. Digit current turn-off at 330 ns completes the memory cycle. The additional time to make up 500 ns is needed to allow the sense amplifier to recover from the digit write noise. Typical sense readout signals from the memory stack are shown in Figure 10.

The memory stack

Each memory frame contains 1024 words, 208 bits long. Four of these frames are interconnected at the logic level. The associated circuits which these four frames share are packaged in a configuration similar to a memory frame. The interconnection is as shown in Figure 8. The frames slide into articulating connectors which are mounted on interconnecting controlled-characteristic-impedance multilayer printed-circuit boards. By opening an articulating connector at both top and bottom, any frame can be slid out for maintenance or replacement.

CONCLUSION

Fabrication, assembly, and operation of these half-microsecond memories has proven that large numbers of reliable film substrates are producible and that the completed memories can compete in both speed and price with the high-speed 2-1/2 D-type core memories. The future for planar films looks very bright; both larger and faster memories are in the design stage. These memories will combine the economic advantages of batch fabrication with the fast switching properties of thin-films.

ACKNOWLEDGMENT

The authors wish to express their appreciation to the many people who were responsible for the successful completion of this project—especially to J. T. Lynch and V. Z. Smith for project management; to R. Benn, E. Trimbur, J. Engelman, and A. Hardwick for film production; to A. Bates, W. Wikiera, and L. Fiore for electronics design; and to E. Duckinfield and R. Saunders for mechanical design.

REFERENCES

1 S A MEDDAUGH K L PEARSON
A 200-nanosecond, thin-film main memory system
AFIPS Conference Proceedings 29 281-292 1966 Fall Joint Computer Conference

2 E E BITTMANN L ARNDT J W HART
A 20-MHz NDRO thin-film memory
To be published in IEEE Transactions-Magnetics

3 E E BITTMANN
A 16K word, 2-Mc magnetic thin-film memory
AFIPS Conference Proceedings vol 26 pp 96-106 1964 Fall Joint Computer Conference

Bit access problems in 2-1/2D 2-wire memories

by PHILIP A. HARDING and MICHAEL W. ROLUND
Bell Telephone Laboratories
Naperville, Illinois

INTRODUCTION

The obvious cost advantage of a 2-wire 2-1/2D core mat over a 3-wire mat has, in the past, been offset by the increased complexity of the access and detection circuitry required for a 2-wire array. This paper will concentrate on 2-wire bit accessing schemes and describe one which appears to be cheaper and less noisy than the conventional bit access which uses a complete matrix per bit. It will then discuss the read-out noise problems. To predict the amplitude of noise a multistate core model similar to J. Reese Brown's[1] will be developed. The paper will then show how the individual core characteristics can be extrapolated to predict overall optimum memory performance.

Extrapolation of 2-1/2 D memories from 2 D memories

The line drawing of a 32,768 word, 24 bits per word, 2-wire memory (Figure 1A) and the extension into two types of 2-1/2D (Figure 1B and Figure 1C) illustrates the derivation of both the typical 2-1/2D configuration[2] and the 2-1/2D configuration proposed in this paper.[3] In the 2-wire scheme, there are as many independent address lines as addresses and as many bit lines as bits per word. The example shown illustrates 32,768 address or word lines and 24 bit lines. Current on a selected address wire fully switches all cores whose signals are readout on the independent bit wires. Coincident currents on the selected address line and the independent bit lines are used to write back independent bit information into the selected word.

The first type of 2-1/2D memory which we call an independent bit matrix organization, segments each of the bit lines (typically 16 segments as shown in Figure 1B); independent selector circuits, one for each bit, are used to simultaneously select a segment for each bit. Since the bit lines have been segmented, the number of word lines can be reduced by a factor equal to the number of segments. The memory is read by

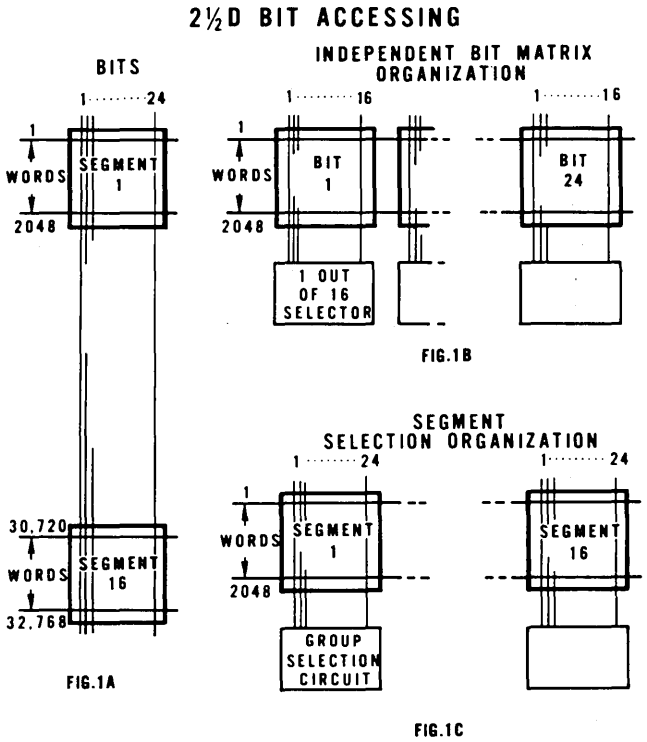


Figure 1—(a) Word organized, 32,768 word by 24-bit memory. (b) Conventional 2-1/2D realization of (a). (c) Segment organized 2-1/2D realization of (a)

sending a word current into the selected word line and simultaneously sending a bit current into the chosen segment of each bit. This achieves a coincident current selection of a core for each bit line. The bit segment selector circuits, although simple in function, are expensive and needlessly repetitious because there is a simultaneous decoding in each of the 24 bits; each selector making the same logical selection. For the memory on Figure 1B, the one out of 16 selector circuit requires at least 8 bidirectional or 16 unidirectional switches per selector and since there is a selector per bit, this scheme will require 192 bidirectional or 384 unidirectional switches.

A second form of 2-1/2D is a simple rearrangement of the segments as shown in Figure 1C. In this case, the adjacent segments of 24 bits are grouped rather than the 16 segments of each bit. The two-dimensional memory section called segment 1 in Figure 1A maps into the segment 1 section of the 2-1/2D memory in Figure 1C; the higher numbered segments map correspondingly. A single group selector circuit is activated and current is simultaneously sent to the 24 lines of the segment associated with that selection circuit. Word current is then sent to a selected word line. The simultaneous word current and bit segment currents select 24 cores in a word for readout. The group selection circuit is obviously a higher power circuit since it must deliver current to 24 lines but it has a much reduced logic decoding. A single one out of 16 selection must be made to pick the proper group selection circuit. This scheme, as will be shown later, can be realized with a total of 8 switches rather than the 192 or 384 in the alternative 2-1/2D memory, without materially affecting other component counts.

Independent bit matrix

A block diagram of the independent bit matrix organization is shown in Figure 2A. There is a word access selector shown at the left. The bit selection circuitry is made up of 24 independent matrices, one of which is schematically depicted in Figure 2B. The load at each matrix crosspoint contains two memory lines connected by a readout transformer, which also balances the bit current. The bit circuitry contains a total of 384 readout transformers, one for each pair of memory lines, 96 bidirectional switches, 192 unidirectional switches, and 768 bit access diodes. In addition, there must be some means of funneling the secondaries of the 16 readout transformers per bit into a readout detector. This could be done by using a low level selection switch per transformer for a total of 384 switches. At the expense of some loss in signal, one can connect several transformer secondaries in series as shown in Figure 2C, and thereby reduce the number of low level selectors required. In practice, it would be undesirable to connect more than four secondaries in series since the signal must then pass through the equivalent of 8192 cores. Therefore, at least 96 low level switches are needed.

Segment oriented bit access

The segment selection organization depicted in Figure 1C can be implemented by the configuration shown in Figure 3. The typical bit matrix of Figure 2B is replaced by an assembly of 16 diode bridge rails, one of which is activated to select a pair of memory lines

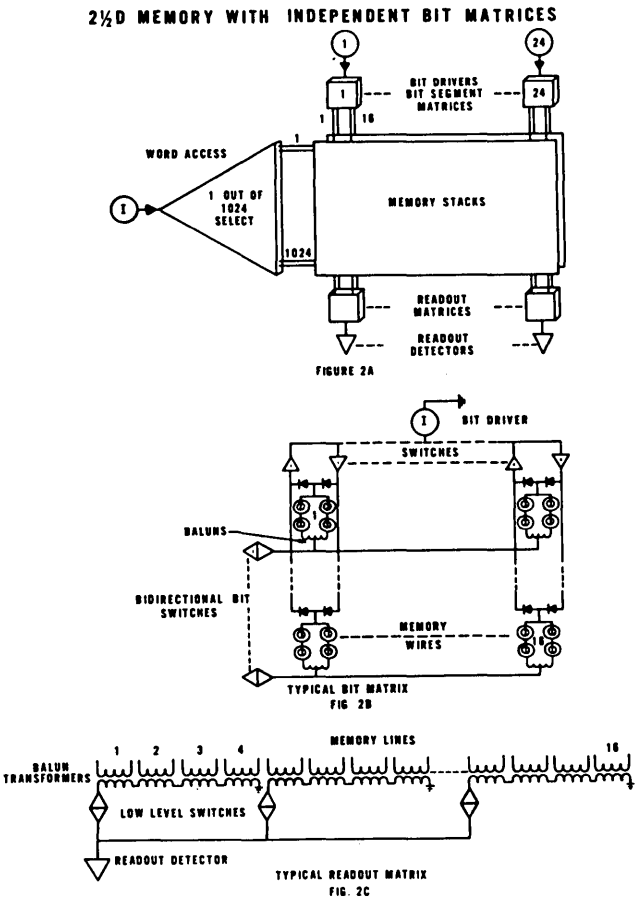


Figure 2—(a) Conventional 2-1/2D memory. (b) Typical bit matrix. (c) Typical readout matrix

for each bit. All of the diodes except for those associated with a selected rail are back biased by the +v and −v potentials shown on the left of Figure 3. This automatically isolates unselected bit lines so that the bit current driver will not send current through these segments and signals generated on the segments will be isolated from the readout transformer at the top.

The selected rail is activated by driving current through the primary (shown in Figure 6) of the transformer associated with the selected rail. This forward biases all diodes on that rail and provides a phantom ground to the lower ends of the pair of memory wires in each bit associated with the selected rail. This provides a current path for the bit driver and a signal path for readout to the transformer at the top.

This technique eliminates the need for low level selectors and requires only 24 readout transformers. For the short bit line segments being considered here, the transmission time of a bit line is short compared to the switching time of a core. Hence the effect of signal reflections due to the 15 open-circuited lines paralleling the selected line is slight. The capacitive

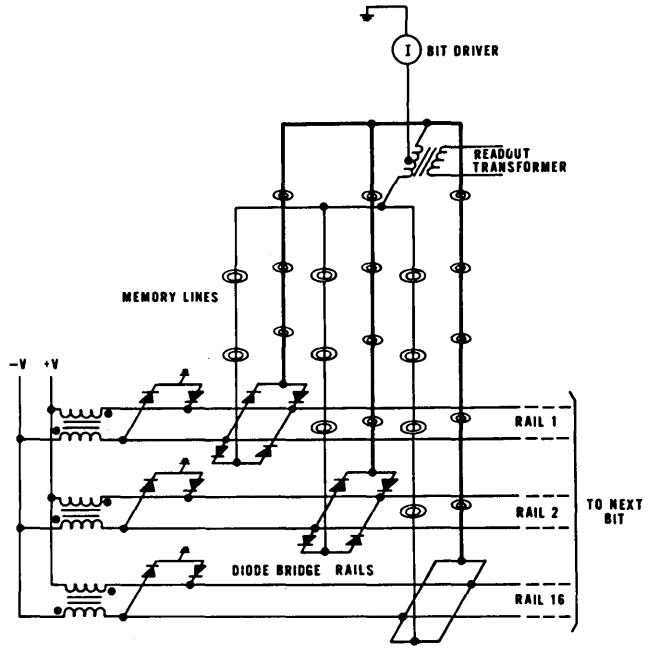


Figure 3—Group selection circuit with inherent readout selection

loading of these 15 lines on the selected line will tend to degrade the current rise time, but this can be minimized by sectioning the access into a multi-dimensional configuration which will be described later.

Alternative organization

There is an alternative organization which utilizes half the number of diodes, but increases the number of readout transformers. This scheme shown in Figure 4 is almost identical to the scheme shown in Figure 3; it contains 16 pairs of rails, but instead of having front and back lines independently connected to their own diode bridges, it couples the front and back segments by means of a balun-readout transformer. The center of the readout transformer is connected to the rails through a bridge diode configuration. The selected rail causes the bit current to flow to the selected front and back bit line while all other segments block current. The readout signals are coupled to the individual bit segment transformers. However, to obtain readout isolation, it is necessary, with this arrangement, to switch the secondaries of the 16 readout transformers by means of low level selectors to a common readout detector circuit.

Multi-dimensional access

The capacitive limitations imposed by paralleling many memory lines can be reduced by using the technique illustrated in Figure 5. The 16 pairs of lines of a given bit are divided into two groups, each of which is connected to a separate readout-balun transformer. The desired pair of bit lines is selected by simultaneously activating one of eight rails at the bottom and either Rail A or Rail B at the top. This method effectively halves the parasitic capacitance seen by the bit driver.

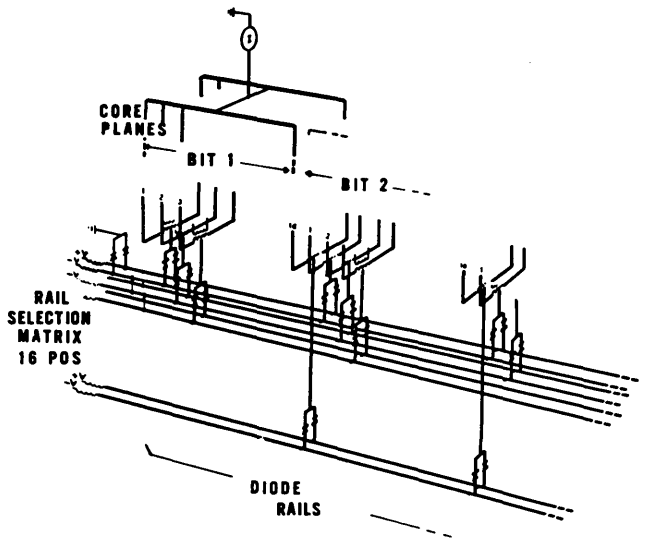


Figure 4—Group selection circuit without inherent readout selection

Rail selection matrix

The rail selection matrix shown in Figure 6 is an electronic 24 pole 16 position switch. At the right of the figure is shown a typical rail, normally back-biased. The rail is activated by current flowing out of the secondary windings of the transformer through the diodes with the grounded bridge node; this forward-biases all diodes of the multiarm bridges on the rail. Thus, each node is virtually grounded through the forward-biased diodes during the entire memory cycle regardless of bit current polarity on the bit segment wires. By alternating the direction of current on adjacent bits (shown by sending bit 1 current up and bit current 2 down during read time in Figure 6), the current handled by the transformer is halved. In effect, the transformer and grounded diodes merely switch the current from a bit to its neighbor.

The transformer primaries are themselves in a 16 crosspoint matrix containing 8 unidirectional 1 ampere

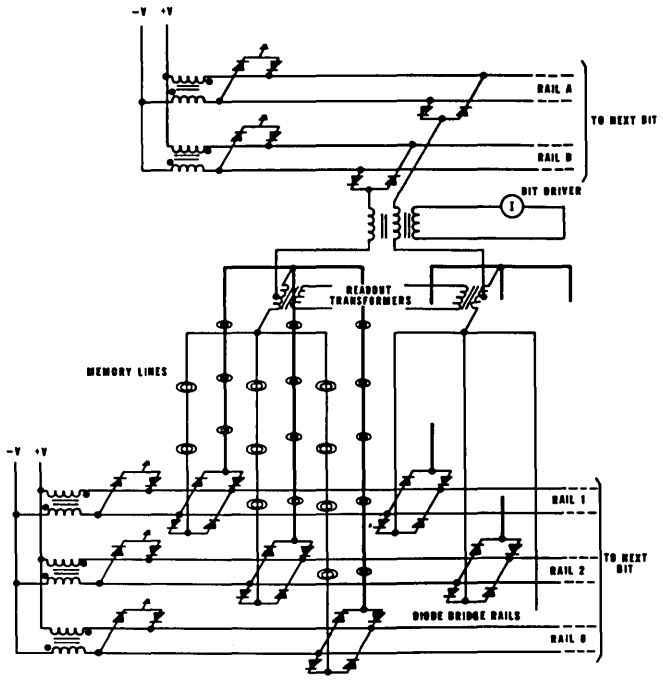


Figure 5 – Multi-dimensional group selection

switches. By selecting one of the four switches at the top and one of the four switches at the bottom, current is sent to one of the 16 crosspoints which, in turn, couples through the transformers to forward-bias a rail. In this drawing, we illustrate 6 transformers per crosspoint. This is done to limit the transformer current and rail current; by subdividing the rails, and cascading transformer primaries, currents in the order of only 1 ampere can drive an entire 24 bit word. In summary, the organization shown in Figure 6 has a number of distinct features. These are:

1. Bit segments are virtually grounded when selected so that no voltage drop is required by the access switch. This lowers the voltage to the bit current drivers and improves current control capability.
2. The selected bridge nodes have very low dynamic impedances because the current paths pass from one bit to its neighbor without having to transfer through long wire lengths with attendant parasitic inductances and capacitances. During the memory read time there will be an insignificant change of current out of the transformer secondaries and negligible net current into the ground of the reference diodes. Dynamically the current will flow down bit 2, through the bridge diodes and back up the bridge diodes into bit 1. The entire matrix, with the exception of the diode

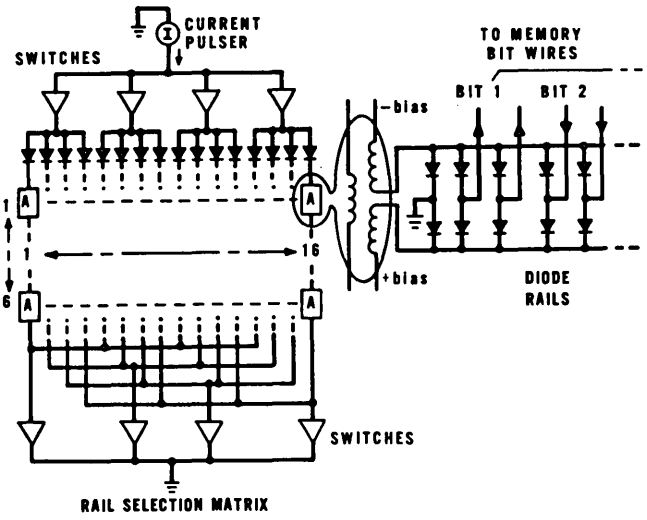


Figure 6 – 24-pole, 16 position bridge drive matrix

rails, can be remotely located from the core module without affecting current wave-shapes in the module.

3. Only 8 switches are required in the primary matrix to simultaneously select 24 pairs of segments out of 384.

A summary of component counts for the various access schemes considered is given in Table I.

Table I – Bit access and detection circuit counts
32,768 Words 24 Bits/Word

	Segment Oriented Access			
	Conventional Access	Without Inherent Readout Selection	With Inherent Readout Selection	Multi-Dimensional
Access Switches	384	8	8	8
Access Diodes	768	960	1728	1752
Access Transformers	0	96	96	60
Readout Transformers	384	384	24	48
Low Level Selectors	96	96	0	0

Influence of the core characteristics upon memory performance

For a 2-1/2D 2-wire configuration the requirements of the core differ from the standard three dimen-

sional coincident current core. It is obvious that the memory output signal is heavily influenced by the noise generated by bit current shuttling the many cores of the bit segment wire. The readout transformer must measure a small core switching signal in the face of thousands of cores shuttled by half currents. This is an order of magnitude larger than the typical three dimensional memory, where the readout must be sensed in the face of only a few hundred cores shuttled by X and Y half currents.

To analyze the bit current noise, we make use of the 6 state hysteresis loop characteristic of the core shown in Figure 7. As will be seen later, it is desirable to operate the 2-1/2D memory with offset currents; i.e., bit current less than word current. For this case, a number of additional intermediate states exist. However, the worst case disturb condition is generated by the larger amplitude word current and we can neglect the less noisy intermediate states. The states are defined as follows:

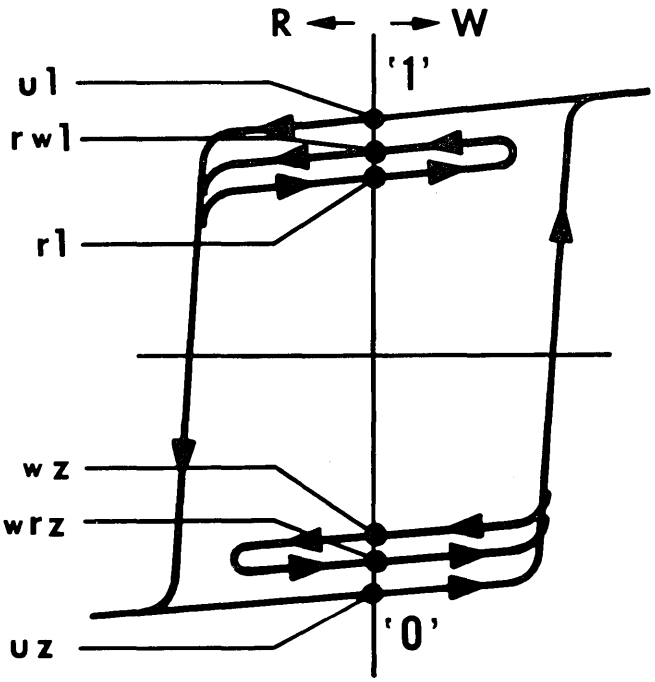


Figure 7—6 state hysteresis loop model

1. ul is the state arrived at when the core is switched by a full write current.
2. rl is the state arrived at when a core in ul is excited by a half read current. There is another state somewhat lower than rl called dl which will occur when the core is continuously exicted by half read currents, however, the core outputs are almost identical in states rl and dl.
3. The state rwl occurs when a core in rl or dl is excited by a half write current.
4. Subsequent excitations of half read and half write currents shuttle the core between rl and rwl.
5. The state uz occurs when a core is fully switched by a read current.
6. The state wz occurs when the core in the uz state is excited by a half write current. Similarly, there is an upper state dz which is achieved when the core is repeatedly excited by a half write current. However, the difference between wz and dz is small as far as the core responses are concerned.
7. The state wrz is achieved when a core in the wz or dz state is excited by a half read current.
8. Subsequent excitations of a half read and half write current shuttles the core between wz and wrz.

A core in the states ul, rwl, wz, and dz exhibit irreversible flux switching when excited by half read currents. Irreversible domain wall motion occurs even for very low half read currents. Thus, there will be an output from cores in these states even after the exciting current rise time has expired. Figure 8 illustrates the effect for half read bit currents. The core in state ul has the uVhl output, rwVhl is the output of the core in state rwl, and wVhz is the output of the core in state wz. All of these responses exhibit a recovery tail due to domain wall motion even after the rise time of the excitation current. Cores in the states rl, dl, wrz, and uz exhibit mostly reversible flux when excited by a half read current; the output is negligible after the current rise time. This is illustrated in Figure 8 by rVhl (rl state), wrVhz (wrz state), and uVhz (uz state).

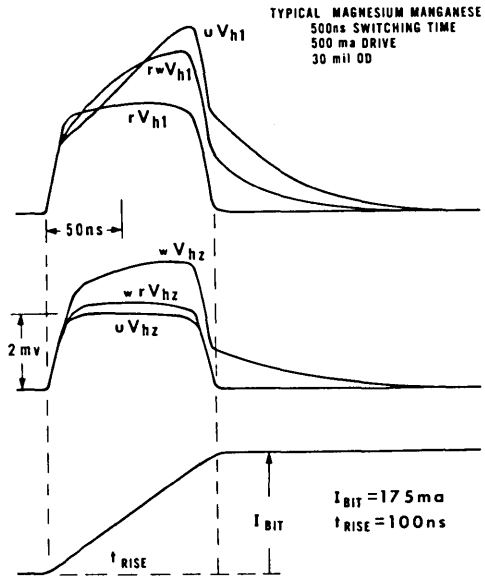


Figure 8—Core output waveshapes when excited by half-select bit current

For the 2-wire 2-1/2D memory shown, the bit half current will excite 1,024 cores in the front plane and 1,024 cores in the back plane whose output signals are sensed differentially in a readout transformer. If the cores in the front plane are in an irreversible flux state while the cores in the back plane are in a reversible flux state, a large recovery tail noise will occur. It will exist long after the bit rise time is over and will interfere with the signal output. In a 2-1/2D memory with a read regenerate or erase and write cycle, it is impossible to have any cores in the uz state and no more than one core can be in the ul state. Therefore, for practical purposes, the worst case will occur when all positive sensed cores are in the rwl state and negative sensed cores are in the rl or wrz state or vice versa.

Word access wiring

To improve memory performance, there is an obvious desire to minimize the number of cores in the front plane or the back plane that are in the irreversible flux state. Fortunately, there is a word access wiring technique that can reduce the number of irreversible flux state cores by a factor of two; the technique is illustrated in Figure 9. A typical bit loop with a readout transformer at the top, the bit drive current at the center of the readout transformer, and the bridge diode rails at the bottom are shown in Figure 9A. The bit drive current splits evenly between the front and back segments into the grounded nodes at the bridge diode. The transformer at the top not only senses the switching core output, but equalizes the current on the front and back plane because of its balun connection. However, the noise difference between the front and back cores due to bit current will show up as a differential signal across the readout transformer.

By wiring the word loops so that a word loop intersects two adjacent cores on the same bit segment and by separating front plane word loops from back word loops four distinct advantages result.

1. Either the upper or lower core of the pair excited by the word loop can be selected by the direction of word loop current.[4] Thus, the number of word loop accessing circuits are reduced by a factor of two.
2. The upper core will generate a noise that is opposite to the noise of the lower core when excited by word loop current and the readout transformer will see a cancelling signal from two adjacent cores; this will lower the noise due to the word loop current.
3. The most important advantage, however, is the fact that when the word loop current pulses, it automatically drives one of the two cores into the reversible flux state. Thus, it is impossible to have more than half the cores on a bit segment excited by word current in an irreversible flux state. It can be shown that this cuts the bit line recovery tail noise in half.
4. The cores can be oriented in line rather than in a diamond pattern. This allows the cores to be on closer centers.

Figures 9B and 9C illustrate an additional advantage achieved with the word access wiring shown. In high speed memories it is also necessary to reduce the capacitance coupling between the word loop and the bit loop. Unfortunately, the word loop of Figure 9A couples to only one side of the bit loop. Thus, any voltage bounce on the word loop would capacitively couple into one side of the bit loop and would be sensed as a differential signal by the readout transformer. It we arrange the word loop access matrix as shown in Figure 9B so that half the word loops on any rail (vertical or horizontal) couple to the front segments of the bit loop and half to the back segments of the bit loop, then voltage bounces in the word access rails will generate equal capacitance coupling signals on the front and back segments of the bit loop. The readout transformer will not sense these signals; it will reject them as common mode noises voltages. Figure 9C illustrates physically how the word loops should be wired to realize common mode cancellations; the word loops interlace alternately between the front and back plane of the module.

Memory signals

Figure 10 illustrates the bit current excitation and the resulting output signal. The bit current is pulsed first to allow time for the recovery tail to expire. After recovery tail expiration the word current is turned on. A large inductive noise pulse will occur during the rise time of bit current. After rise time, there will be a bit line recovery due to irreversible flux switching. The recovery tail will ride upon a DC pedestal which is due to the difference in resistance and difference in diode voltage drops between the front and back plane. In this memory, therefore, the readout detector must sense the output signals with respect to the pedestal. A number of techniques can be employed such as DC restoration or delay line restoration;[5] the actual scheme utilized is a capacitance charging DC restorer. After restoration, the word current is pulsed to readout a "1" signal or a "0" signal from the selected core. The "0" signal will decrease after the rise time of word current so that during the peak of the "1" signal the major noise present will be the recovery tail due to bit current.

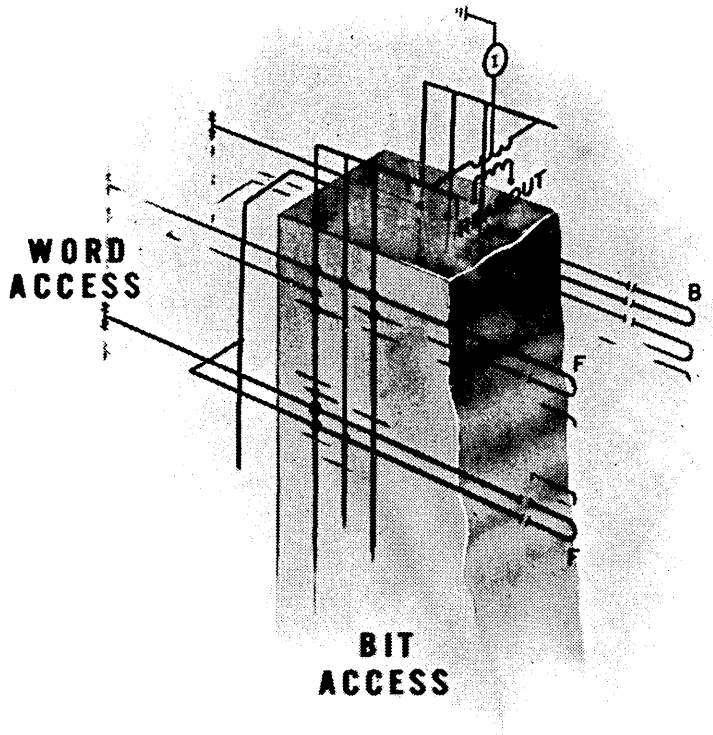
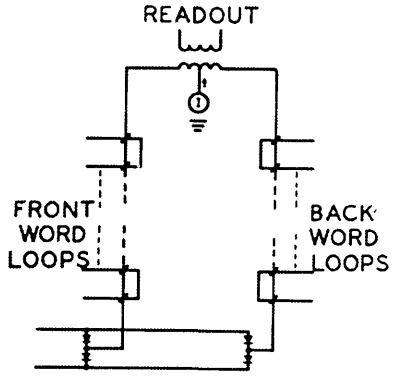
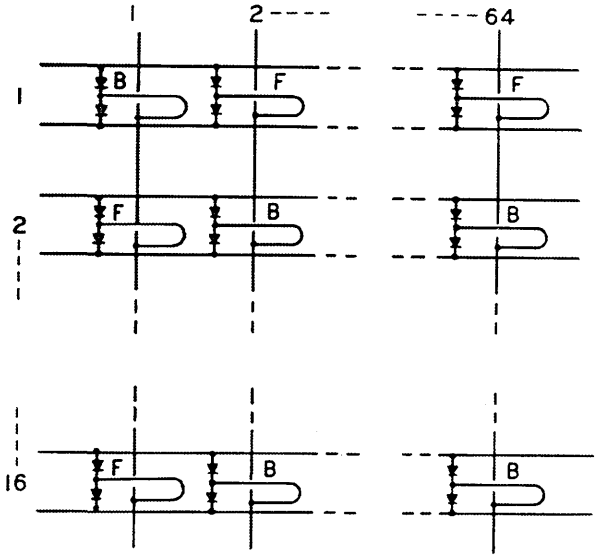


FIG. 9C



BIT LOOP
FIG. 9A



WORD ACCESS
FIG. 9B

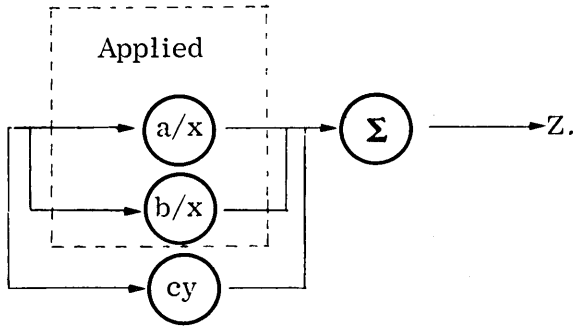


Figure 9—Word access wiring pattern for minimum magnetic and capacitive noise. (a) Bit loop. (b) Word access. (c) Pictorial representation

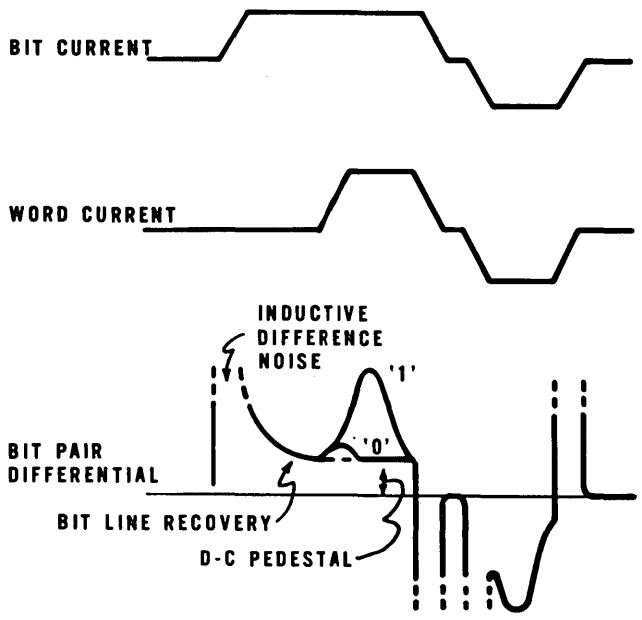


Figure 10—Memory current timing and readout waveform

Thus, it is evident that the signal to noise ratio is predominantly influenced by the bit line recovery tail and not by the "0" readout of the selected core.

To quantitatively prove this fact, we study the core characteristics shown in Figures 11, 12, and 13. The core characteristic in Figure 11A is the millivolt out-

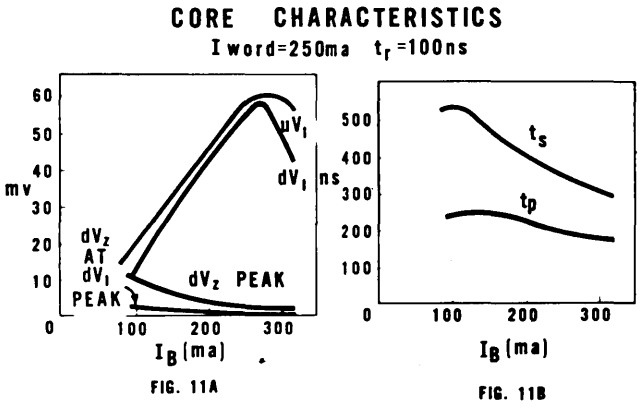


Figure 11—Core output characteristics for current timing as shown in Figure 10. (a) Readout amplitudes. (b) Time intervals.

put of a core switched by the coincidence of a bit current, whose amplitude is varied, and a fixed word current of 250 milliamperes with a 100 nanosecond rise time; the bit current is turned on first and the word current second as shown in Figure 10 to simulate the actual 2-1/2D operation. The output occurs during the word current, but we are plotting the output as a function of bit current amplitude. Two important conclusions can be drawn from this characteristic. First, the plot differs from the typical curves of most manu-

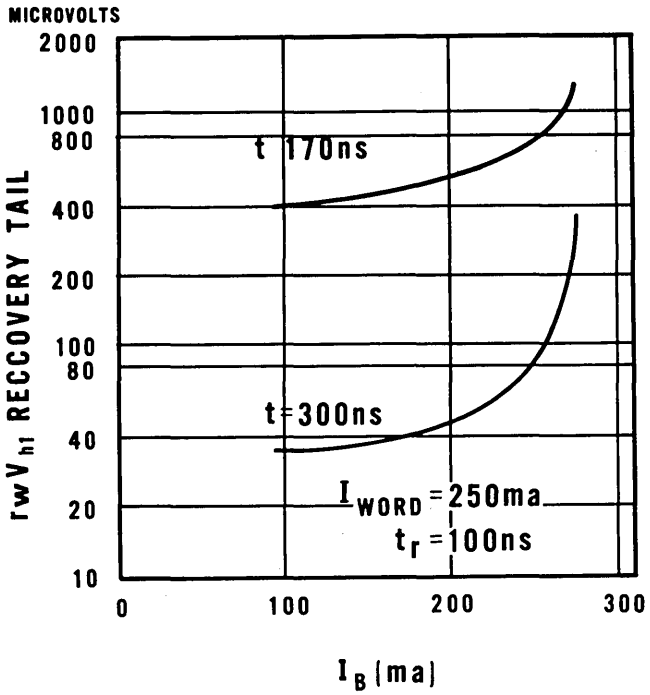


Figure 12—Amplitude of rwV_{h1} recovery tail as a function of I_B at fixed time t after start of I_B

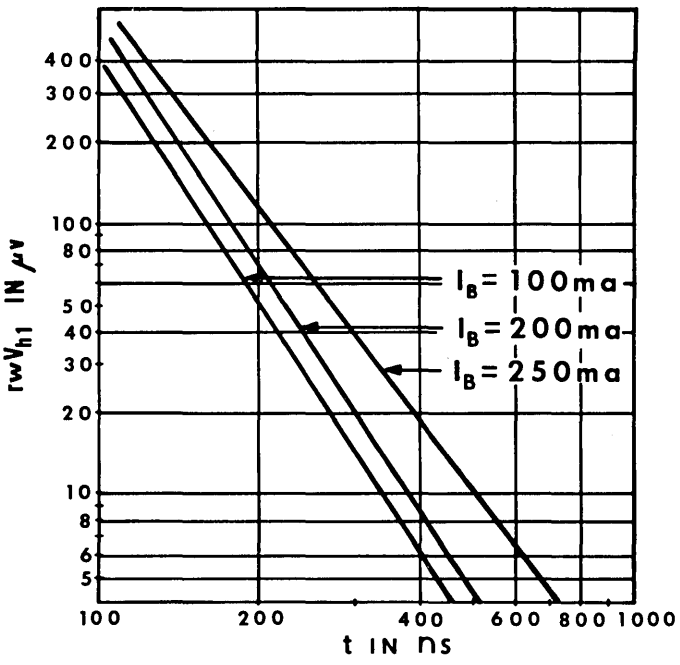


Figure 13—Amplitude of rwV_{h1} recovery tail as a function of time for various values of bit current

facturers because the manufacturers do not characterize the cores with a time staggered sequence of coincident currents. Secondly, the dVz output at the peaking time of the dVl is less than 1% of the "1's"

output even at bit currents of less than 150 milliamperes. Even though the dVl is maximum at 275 milliamperes, there is an obvious tendency to use low bit current to reduce the bit recovery tail; Figure 12 illustrates quantitatively how low bit currents reduce tail noise. A plot of rwVhl versus bit current at 170 nanoseconds and 300 nanoseconds after start of bit current is a measure of the recovery tail for a core since the rise time of the bit current is 100 nanoseconds. For a bit current of over 250 milliamperes, the recovery tail becomes extraordinarily large and the memory noise due to irreversible flux increases rapidly. Even at low bit currents and at a recovery tail time of 300 nanoseconds, an output of 20 millivolts will occur from 500 cores, generating a noise close to 50% of a "1's" output. This proves that the recovery tail is by far the largest noise source and limits the performance of the 2-wire 2-1/2D memory. Figure 11 proves that we need high bit currents to get large dVl outputs up to a current of 275 milliamperes. Figure 12 proves that we need low bit currents to get low recovery tail; therefore, there must be an optimum to achieve best signal to noise ratio.

Figure 13 contains plots of the recovery tail as a function of time on a Log-Log scale for different bit current amplitudes. The straight lines indicate an interesting characteristic; the output voltage falls off approximately as the inverse of time squared ($^1/_{t^2}$) for moderate bit currents. Another important conclusion exhibited by Figure 13 is that the recovery tail to 10% of the nominal core output (for bit lines of 1,024 cores) lasts for the order of 400 to 600 nanoseconds for a core with a switching time in the 400 to 600 nanosecond range. Thus, the recovery tail is as long as the switching time.

Signal to noise optimization

By combining the dVl output of Figure 11A, the switching time, ts, (initial 10% to final 10% of dVl) and peaking time, tp, (initial 10% to peak of dVl) characteristic of Figure 11B and the recovery tail voltage of Figure 13, it is possible to obtain a signal to noise plot as a function of bit current for the 2-1/2D memory. Figure 14 contains a plot of the signal (dVl) to noise (Recovery tail due to the bit current) for 1000 cores on a bit line as a function of bit current, with memory cycle time as constraint. The memory cycle time is defined as the read switching time plus the write switching time of the core plus the recovery tail time. This plot has been carried out for a typical 500 nanosecond, 500 milliampere, 30 mil outer diameter core in a memory cycle time of 1.4 microseconds, 1.2 microseconds, and 1.0 microseconds. The result is that the best signal to noise ratio occurs at a bit

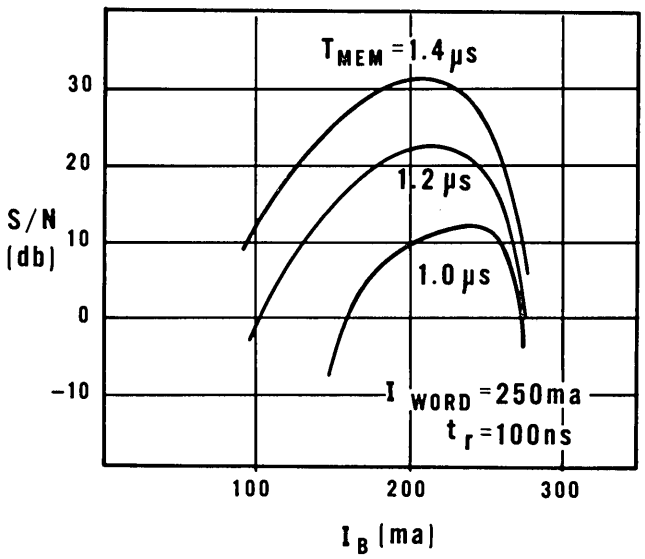


Figure 14—Ratio of dV_1 to amplitude of rwV_{hl} tail at $t = t_t$ such that $T_{mem} = t_t + 2t_s$

current of 225 milliamperes; this is optimum at a bit current 25 milliamperes lower than the word current and 50 milliamperes lower than the peak "1's" output current. Similar data have been taken for many cores with almost the identical result. A converse plot, for a fixed signal to noise ratio, of memory cycle time versus bit current proves that minimum cycle time will occur at a bit current of 225 milliamperes. Thus, we can conclude that the best signal to noise performance and the minimum cycle time will occur at a bit current lower than the word current and underdriven to the extent of decreased "1's" output.

SUMMARY

In this paper, we have described a class of new access schemes that appear to be economic without degrading performance. An analysis of secondary and primary core characteristics has been carried out to assist in predicting optimum memory performance.

REFERENCES

1 J R BROWN JR.
First and second order ferrite memory core characteristics and their relationship to system performance
IEEE Transactions on Electronic Computers Vol EC-15 No 4 August 1966 pp 485-501

2 T J GILLIGAN
2-1/2D high speed memory system—past, present, and future
IEEE Transactions on Electronic Computers Vol EC-15 No 4 August 1966 pp 475-485

3 P A HARDING M W ROLUND
Novel low cost design for 2-1/2 D storage systems

1967 Solid State Circuits Conference Digest of Technical Papers Vol X – IEEE Cat No 4C49 pp 82-83

4 T J GILLIGAN P B PERSONS
High speed ferrite 2-1/2 memory
AFIPS Conference Proceedings FJCC Vol 27 Washington D C Spartan Books 1965 pp 1011-1021

5 A M PATEL J W SUMILAS
A 2.5D ferrite memory sense amplifier
1966 International Solid-State Circuits Conference Digest of Technical Papers Vol IX IEEE Cat No 4C27 pp 96-97

Engineering design of a mass random access plated wire memory

by C. F. CHONG, R. MOSENKIS, and D. K. HANSON
Univac Division, Sperry Rand Corporation
Philadelphia, Pennsylvania

INTRODUCTION

Among the newer memory elements, plated wire has been shown to be a serious contender for aerospace and central store applications.[1,2,3] This paper describes a memory development project, sponsored by the Rome Air Development Center, to extend the application of plated wire into the area of mass storage.

The basic memory module consists of 10^7 bits; the mechanical package can hold 10 modules. The potential speed is a 1-to-2-microsecond word rate. Preliminary system, stack, and circuit designs have been completed and a partially loaded model was fabricated and tested.

Memory organization

The memory under development has a capacity of 10^8 bits. This capacity is achieved by stacking ten 10^7 -bit modules into one unit. Figure 1 shows the arrangement and organization of such a memory Each module has its own set of driving circuits and sense amplifiers. This arrangement leads to a fast, random-access memory, readily realizable mechanically; it is justified from the viewpoint of modularity and cost because the electronic circuits are shared by a large number of bits. All modules share one set of auxiliary circuits, which include the address decoders, timing circuits, information registers, and power supplies.

The organization of the 10^7-bit memory module is shown in Figure 2. The memory plane contains 2048 word lines and 4608 plated wires. The word lines are spaced at 0.045-inch centers and the bit lines are spaced at 0.015-inch centers. This results in a storage density of approximately 1500 bits per square inch. The reason these spacings were selected will be discussed later.

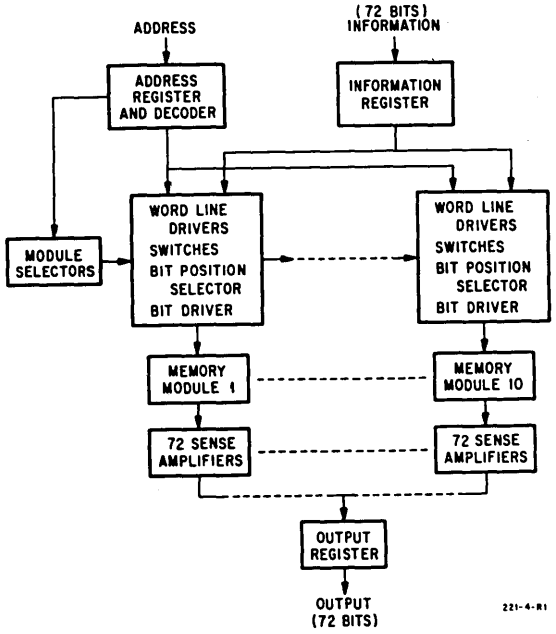


Figure 1 – Arrangement and organization of 10^8-bit memory

The plated wire used is a nondestructive readout (NDRO) element with equal word currents for reading and writing.[3] This property makes it unnecessary to have rewrite circuitry for each stored bit. A word line may be made many machine words in length; each time all the bits in such a word line are interrogated, only the bits belonging to the selected word are routed by a set of gates, called the bit-sense matrix, to the sense amplifiers. After interrogation, all the originally stored information at each bit location along the word line is left unchanged. Correspondingly, the same set of bit-sense matrix gates is employed to route the bit drivers to the proper bit lines of the memory. This feature is illustrated in Figure 3. This property is very

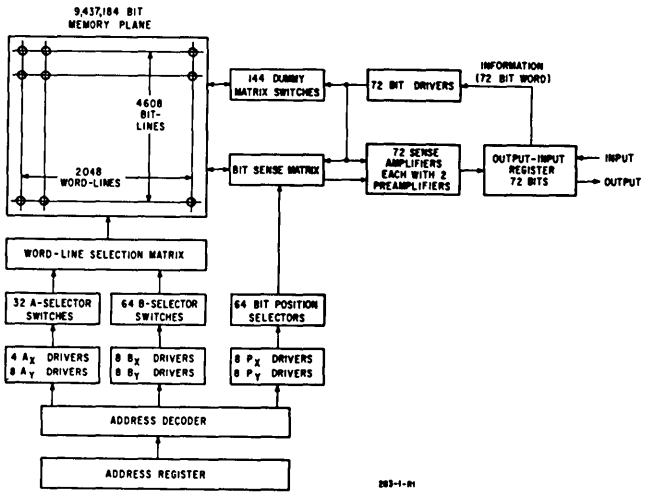


Figure 2 – Organization of 10^7-bit memory module

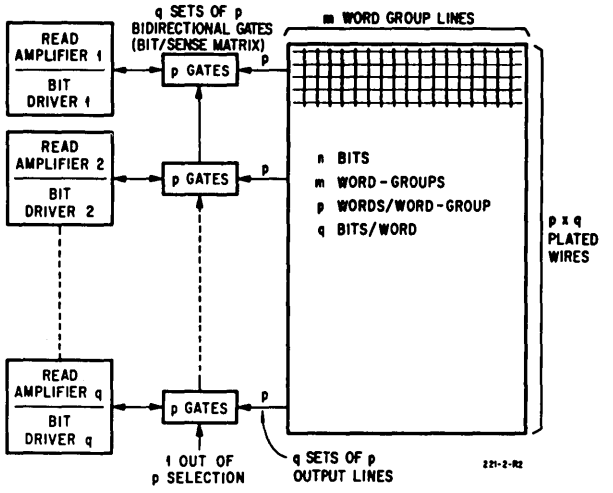


Figure 3 – Modified word-select memory organization

important because it allows a memory configuration to be chosen which leads to a minimal number of bit and word drivers and sense amplifiers.

The word-line matrix, bit-selection matrix, and selector switches are included physically in the memory stack; as a result, the number of connections between the memory-access circuitry and the module is reduced to a minimum.

Stack design

Two possible word-line constructions were considered. The first, called a half-turn line, consists of a single flat conductor over the plated wire, the wire being over a ground plane. The second type, a full-turn line, is a flat conductor wrapped around the wire, with the entire assembly mounted over a ground. The full-turn line provides a stronger and more confined field per unit of word current, thus giving less adjacent-word interaction and allowing lower word currents to be used. However, it presents significant mechanical problems in plane construction. Word lines must be etched double-length, and accurate registration between top and bottom conductors must be maintained.

Work has been published[4] which indicates that the placing of a magnetic keeper over the half-turn word line gives that configuration the advantages of the full-turn line, namely, low operating word current and low susceptibility to adjacent word interaction. Tests were performed to verify the theory; the results are shown in Figures 4 and 5 for full-turn and keepered half-turn word lines on 45-mil centers. In these figures, I_{wP} is the so-called "pop point" word current, representing a lower limit on word current; I_{wD} is the word current for destructive readout, and I_{wA} is the current flowing in the adjacent word line which will cause significant interaction in the bit under test. Based on these tests, it was decided to use a half-turn keepered word line, 33 mils wide, on 45-mil centers.

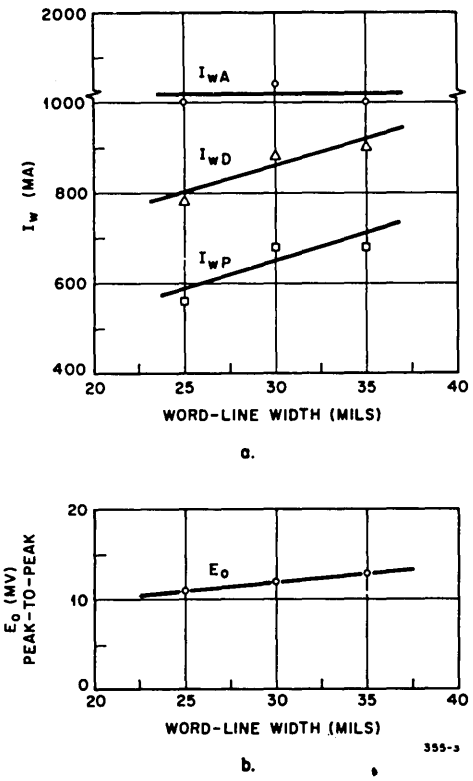


Figure 4 – Word current (a) and output (b) as functions of line width, for one-turn copper word lines

When an 8-foot sense line was contemplated, crosstalk between plated wires during a read operation was a major consideration. With a readout signal being

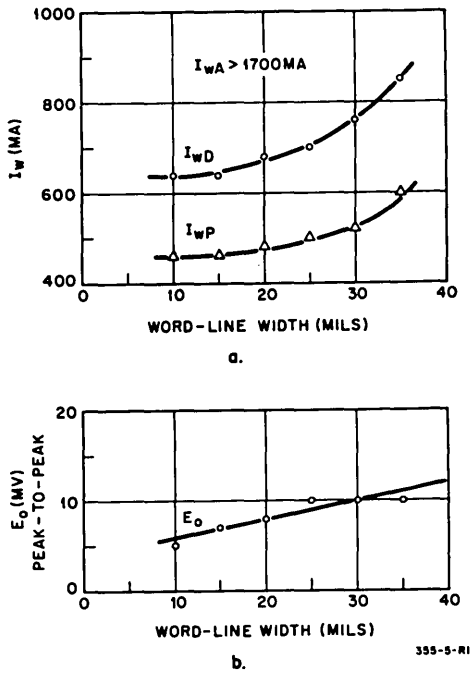


Figure 5—Word current (a) and output (b) as functions of line width, for half-turn copper word lines with mu-metal keepers

generated on all wires crossed by a word line, portions of the adjacent wire signals couple across to the wire being sensed. This crosstalk will either add to or subtract from the desired signal. Crosstalk signals have been broken into their forward and backward components, according to Feller et al.,[5] with appropriate modifications to conform to the fact that one end of the plated wire is grounded, while the other is properly terminated by the bit-sense matrix. It was found that for the word-current rise times anticipated, the peak of the crosstalk from adjacent wires could be more than 50 percent of the peak signal being sensed. However, the time integra of the crosstalk is zero to a first approximation. Therefore, it was decided to use an integrating sense amplifier for the memory to eliminate the effects of crosstalk.

Figure 6 shows a diagram of the plated wire selection scheme. Two dummy wires are associated with every 64 plated wires. Dummy line A is selected any time a plated wire from the 1-to-32 group is selected. Dummy line B is likewise selected at the same time as a wire from the 33-to-64 group. The dummy wires are used for noise cancellation. They are nonmagnetic wires and do not switch. The bit line selection scheme for a bit group is shown in Figure 6.

When information is written into the memory, bit current is driven down the selected wire and down a dummy wire in order to minimize the effect of a large bit-transient voltage in the differential sense amplifier, which would occur if current flowed only in the plated wire. When information is read from the memory, identical noise is coupled into the plated wire and the dummy by the word current and hence can be rejected by the differential sense amplifier.

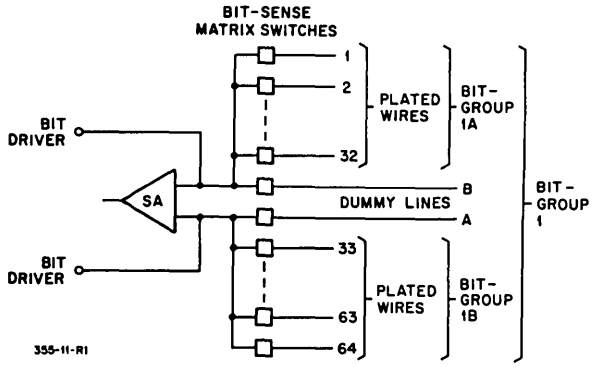


Figure 6—Plated-wire selection scheme

Because the dummy wire associated with bit-group 1A is selected when bit lines in group 1B are being interrogated, and vice-versa, a symmetrical situation is created which reduces the noise in the sense amplifier due to capacitance coupling of word line noise through the unselected bit-sense matrix switches. To explain further, if all the plated wires were connected to one differential input of the sense amplifier and only the dummy wire to the other differential input, noise coupled to the sense amplifier through the off-impedance of the bit-sense matrix switches would cause a noise imbalance. The symmetrical assignment of plated wires and dummy wires allows these off-impedance coupled noises to be in balance and to be rejected by the differential amplifier.

Stack construction

The 10^7 -bit module consists of two half-planes, one 4 feet 9 inches × 3 feet 2 inches, the other 4 feet 9 inches × 3 feet 9 inches, joined by a hinge along their 4-foot, 9-inch sides (see Figure 7). Forming the core of the plane is a 0.25-inch aluminum honeycomb structure, covered with a 5-mil aluminum skin which serves as a ground. Both surfaces of each plane are used. Tunnels to contain the plated wires are made in continuous lengths, 252 wires wide, as follows: A sandwich is formed by pressing 8-mil monel wires on 15-mil centers between an 8-mil sheet of Teflon and two outer 2-mil sheets of Kapton film. In the process, the wires become embedded in the Teflon. The material is then cut to the proper length and the monel wires pulled out. The tunnels

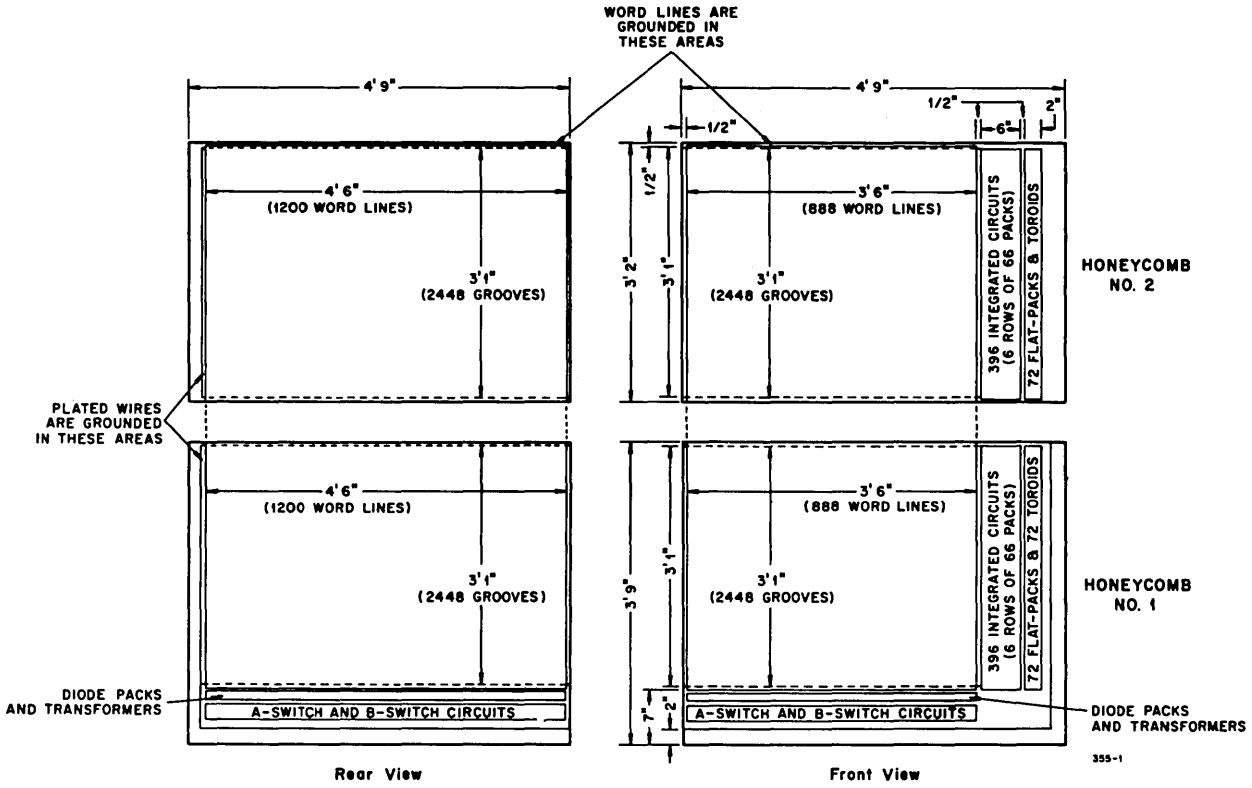


Figure 7 – 10^7-bit module layout

thus formed easily accommodate the 5-mil plated wires. Strips of this tunnel structure are bonded to the aluminum planes with their lengths parallel to the long edge of the plane. The word lines are fabricated as etched cables 120 lines wide of 1-ounce copper on a Kapton film base. They are bonded perpendicular to the tunnels and are continuous across the hinge connecting the two physical halves of the plane. Strips of magnetic keeper material 6 inches wide are bonded over the word lines parallel to them. Small gaps were left between adjacent strips so that noise could not propagate along the keeper for more than 6 inches in the bit direction. Plated wires are connected in series from the front to the back side of the plane and grounded at the far end. At the near end, the wires are terminated in boards containing the bit-sense matrix packages. Other boards, containing matrix diodes and transformers, mount on the plane and connect to the word lines.

Memory circuits

In general, the memory circuits used for this model are straightforward or have been described previously.[3] A description of several unique circuits follows.

Bit driver

The bit driver is shown in Figure 8. The phase-modulated-write method used in this memory system requires the writing of the complement information first, followed by the writing of true information during the writing of a bit. Therefore, if a 1 is to be written in a bit location, a 0 is written first, followed by the writing of the 1. Transistors Q1 and Q2 (see Figure 8) are pulsed sequentially on and off by the information generator in the memory exerciser during a write-1 instruction. This generates a negative and then a positive current in the secondary windings of the transformer. The sequence of pulsing Q1 and Q2 is reversed for a write-0 instruction.

With a nominal collector-supply voltage (V_{cc}) of + 12 volts, resistor R1 limits the current in the primary winding to 64 milliamperes. This allows the secondary to deliver 32 milliamperes to the plated wires and 32 milliamperes to the dummy wires. This arrangement of the bit current prevents the injection of a large, differential, bit-transient signal in the sense amplifier, thereby minimizing the recovery time. This arrangement of the bit driver allows a cycle time of a few microseconds. The transformer consists of two 6-turn bifilar windings located on

opposite sides of the core corresponding to the primary and secondary windings. These windings are intraconnected to produce the proper phase.

Use of a single supply and a single resistor to control the amplitude of both polarities of the bit current ensures their equality.

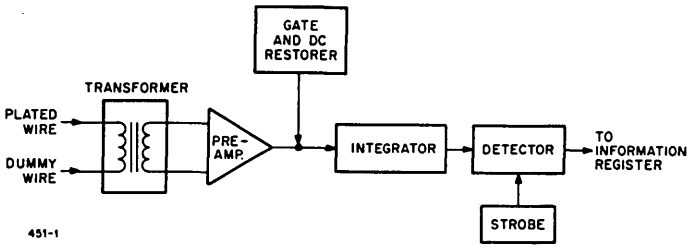


Figure 8 – Bit-driver design

Sense amplifier

An integrating sense amplifier is used for the memory. This detects the time integral of its input rather than its peak amplitude. Such an amplifying system has several advantages in a large memory, including the following:

a. Crosstalk from readout signals in adjacent wires is cancelled, since its time integral is zero. This benefit has been discussed above.
b. Word line noise coupled to the plated wire has, to a first order, a zero time integral. It, too, is cancelled out in the amplifier.
c. Dependence on the fall time of the word current is minimized. Since the area of the plated-wire readout signal is a function of word current amplitude but not of speed, use of an integrating sense amplifier relaxes the requirements on the word circuitry.
d. Approximately equal output areas result from readouts at all positions along the plated wire. A readout from the grounded end of the wire yields a single pulse, while transmission line behavior causes the readout from the other end of the wire to reach the sense amplifier as two like-polarity pulses of lower amplitude. If line attenuation is neglected, the total area of both readouts is the same, so that the integrating amplifier gives identical outputs.
e. Strobe timing is less critical with an integrating sense amplifier. A peak-detecting amplifier must have as narrow a strobe as possible to prevent its triggering on noise. With the signal delays along an 8-foot bit line, narrow strobing would be difficult. The integrator, however, holds its maximum output sufficiently long to permit a wide strobe and looser tolerances on its timing.

A block diagram of the sense amplifier is shown in Figure 9. The transformer input to the preamplifier is required to allow the proper d-c biasing of the preamplifier. It also provides common-mode noise rejection. The gate is a low-impedance shunt which is normally closed. This provides a d-c zero reference at the integrator output. During the interval when the wire signal is to be sensed, this gate is opened. The integrator then charges up, and the gate is closed. Next, the strobe is energized, and the detector provides an output pulse for one polarity of plated-wire signal and no pulse for the other polarity.

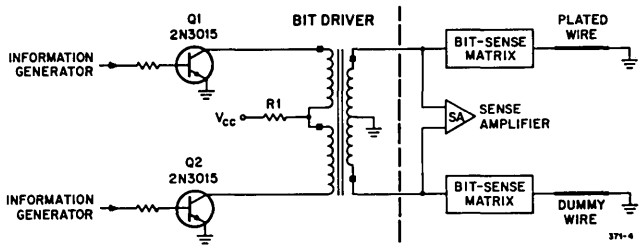


Figure 9 – Block diagram of sense amplifier

A complete schematic of the sense amplifier is shown in Figure 10. As can be seen, the major components are off-the-shelf integrated circuits. To achieve versatility, these circuits generally omit input biasing networks. Coupling capacitors must also be added. The transistor which serves as the gate is operated as a chopper in the inverted mode. This technique provides very low impedance to ground when the transistor is on and could not have been achieved with a standard integrated circuit. The strobe circuit, while available as an integrated circuit, was more readily built from discrete components.

A Texas Instruments SN 5510 serves as the preamplifier. It has a bandwidth of 40 megacycles, more than adequate for the purpose, and a voltage gain of about 80. Only one-half of its differential output is used. A feedback capacitor was added to the RCA CA-3002 linear amplifier to convert it to an integrator. The Fairchild μA 711 is a high-gain amplifier with strobe provisions. It has a very small linear range, about 3 millivolts at the input, and is thus useful as a pulse shaper. The bias adjustment is used to set the desired trigger level. When the signal input exceeds this level, the output follows the strobe pulse.

Tests of the sense amplifier indicate that it achieves all design goals.

Word circuitry

With the two-dimensional diode matrixing used to select word lines in previous memories,[3] half-selected

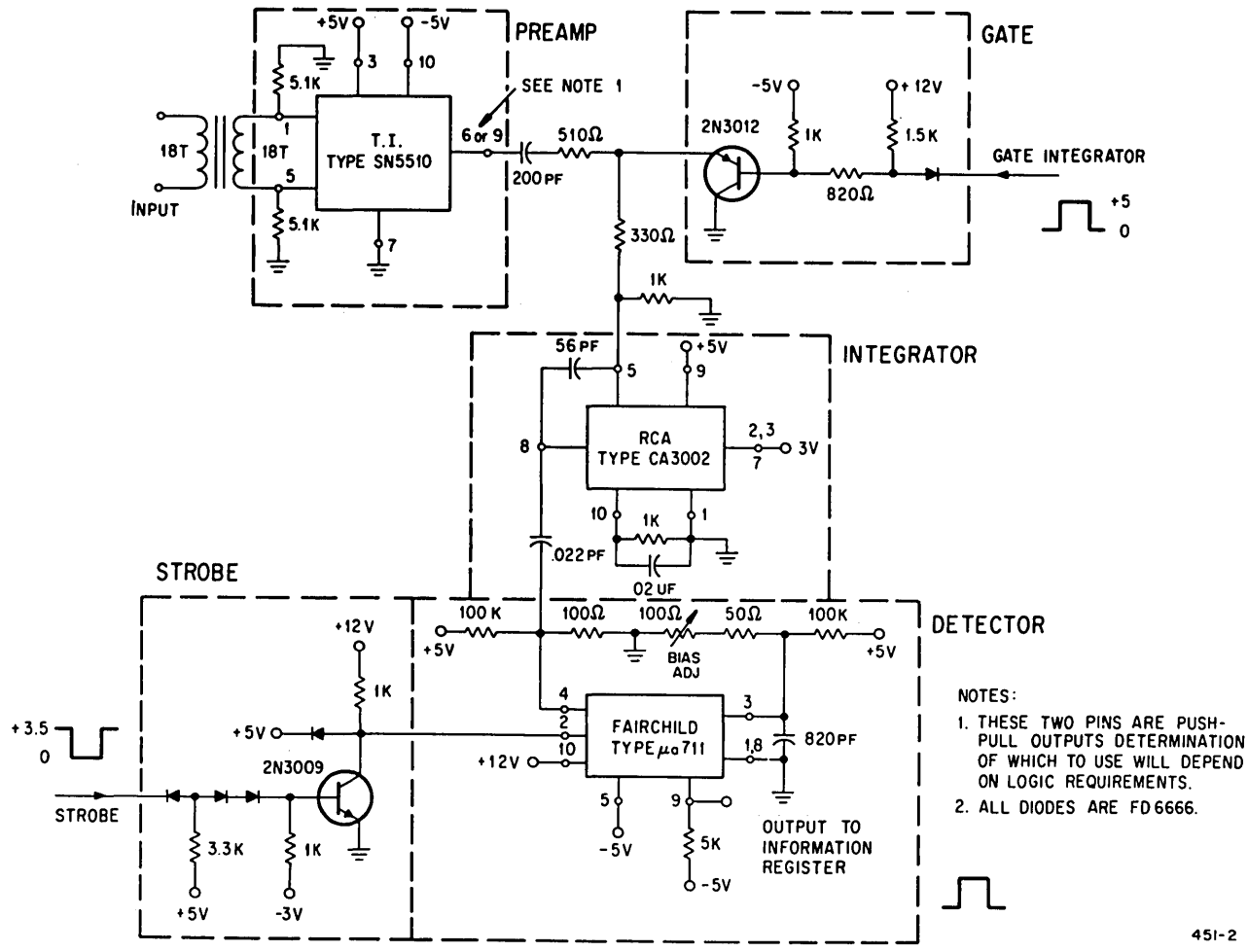


Figure 10—Sense amplifier, schematic diagram

lines were driven through a large voltage excursion by the B-switch. To drive the 0.016-microfarad capacitance of 32 six-foot lines in the present memory would have presented serious circuit difficulties; it might result in a limitation of cycle time and additional system noise. Therefore, a transformer-diode matrix was used instead, with one transformer and one diode per word line, as shown in Figure 11. Conventional diode matrixing is performed on the two ends of each transformer primary just as it had previously been done on the two ends of each word line. By permitting one end of the word lines to be at d-c ground, the transformer-diode matrix reduces system noise.

Memory model

The memory model constructed consists of a mechanically complete 10^7-bit stack. Word line and bit-sense-matrix circuits have been provided to address 32,768 four-bit words at the intersections of 512 word lines and 256 plated wires. These addressable locations were chosen at the four corners of the stack to provide limit conditions. Each addressable word line contains 64 four-bit words, with the remaining tunnels containing nonaddressable plated wires to give a worst-case loading condition of the word line. Two bits of each word are at the driven end of the word line, while the other two are at the grounded end. Similarly, each addressable plated wire passes under 256 addressable word lines at its grounded end and another 256 at its bit-sense-matrix end. Thus, the four corners of the stack are addressed and tested by the memory exerciser described below.

The model was tested at a 200-kilocycle-per-second word rate. The speed was limited by the recovery of a transformer in the circuit driving the bit-sense matrix. Since this limitation is not fundamental to the stack or system design, and since the development specifications call for only a 20-kilocycle-per-second word rate, this circuit was not redesigned.

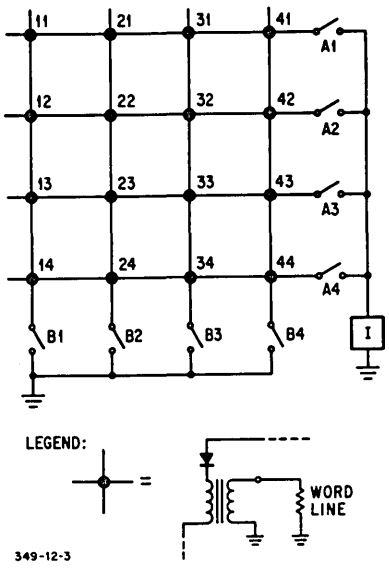


Figure 11 – Word-line selection matrix

Memory exerciser

Operation modes

The exerciser is capable of supplying various information patterns and modes of operation. These modes are write (W), read (R), write/read (W/R), write/read/subscan (W/R/SS). The W and R modes simply allow information to be written into or read from the memory. The W/R mode allows the exerciser to exercise each address (four bits) by writing information into the bit position and immediately reading it out a predetermined number of times. The W/R/SS mode permits the exerciser to write information into all bit positions once, until the memory is filled, and then allows the information to be consecutively read out a preset number of times.

The simplest information pattern that can be checked is all 1's or all 0's. Also, the information can be varied so that each bit contains the opposite information from the preceding bit, which will produce a 1-0 test pattern either down each word line, down each bit line (plated wire), or down both word and bit lines to produce a checkerboard pattern throughout the memory plane.

Worst-case test patterns

The memory exerciser is capable of generating several information patterns. These patterns have been designed with worst-case plated-wire tests in mind.

One test is designed to ensure the nondestructive readout property of the plated wire. This test is implemented by writing into the memory certain information patterns and then repeatedly reading the memory without rewriting. This test is also implemented in combination with other tests.

Another test is to check adjacent-bit disturb. The plated wire, being a continuous-storage medium, is susceptible to adjacent-bit-disturb phenomena. This is tested by writing 1's once into alternate bits in the memory and then writing 0's repeatedly into all other bits. The 1's are then read out and checked. This information is then complemented, the 0's read out and checked, and then the roles of the disturbing bits and test bits are reversed and the test procedure is repeated. An added feature of this test allows reading the test bits repeatedly while writing into the disturbing bits. This procedure then checks NDRO as well as adjacent-bit-disturb effects.

The plated wire is subject to disturbing due to adjacent-wire coupling. This effect is tested by writing into adjacent wires the same information as that stored in the bit under test. The information in the bit under test is then read out. Next, information opposite to that stored in the bit under test is written into the adjacent wires, and again the bit under test is read out. This last test is also used to check the diminution of the output signal from the test bit, as seen by the sense amplifier, by the information of opposite polarity stored in the adjacent wires feeding through the off-impedance of nonselected bit-sense-matrix switches.

CONCLUSIONS

Compatible system design, mechanical fabrication, and unique circuit designs of a mass random access plated wire store have been described. The system design fully utilizes the NDRO property of the plated wire and results in a low electronic component count. Ease of fabrication has been emphasized in the memory stack design. These factors, together with the low plated wire element cost, make an inexpensive mass plated wire store a distinct possibility.

ACKNOWLEDGMENT

The work reported here was sponsored by the Rome Air Development Center, Griffiss Air Force Base, New York, under Contract AF 30(602)3825. The authors wish to thank Mr. Robert Long, the cognizant project engineer, for his cooperation. The assistance of Dr. Joseph Mathias and Messrs. W. Bartik, G. Reid, and A. Schultz, of Univac, is also gratefully acknowledged.

REFERENCES

1 G A FEDDE G H GUTTROFF
A reliable very low power plated wire spacecraft memory
Proc. National Electronics Conference vol 20 pp 681-686 1964

2 G A FEDDE
Design of a 1.5 million bit plated wire memory
Proc. Eleventh Annual Conference on Magnetism and Magnetic Materials J Appl Phys 37 pp 1373-1375 1966

3 J P McCALLISTER C F CHONG
A 500-nanosecond main computer memory utilizing plated wire elements
Proc. Fall Joint Computer Conference 1966 pp 305-314

4 N PRYWES ed.
Amplifier and memory devices: with films and diodes
McGraw Hill Book Company Inc New York 1965 pp 265-283

5 A FELLER H R KAUPP J J DIGIACOMO
Crosstalk and reflections in high speed digital systems
Proc. Fall Joint Computer Conference 1965 pp 511-525

BIBLIOGRAPHY

M BIENHOFF J CAMARATA M SHERMAN
Some considerations in the design of plated wire memory considerations in the design of plated wire memory systems
Proc. IEEE National Symposium on Batch Fabrication Apr 1965 pp 88-102

I DANYLCHIK A J PERNESKI M W SAGAL
Plated wire magnetic film memories
Proc Intermag Conference Apr 1964 pp 5-4-1 to 5-4-6

T R FINCH S WAABEN
High-speed DRO plated-wire memory system
Proc Intermag Conference Apr 1966 paper 12.3

T R LONG
Electrodeposited memory elements for a nondestructive memory
J Appl Physics vol 31 p 123S 1960

S OSHIMA K FUTAMI T KAMIBAYASHI
The plated wire memory matrix
Proc Intermag Conference Apr 1964 pp 5-1-1 to 5-1-6

A V POHM E N MITCHELL
Magnetic film memories, a survey
I R E Trans on Electronic Computers EC-9 p 308 1960

A new technique for removable media, read-only memories

by ROBERT E. CHAPMAN and MATTHEW J. FISHER
U.S. Army Electronics Command
Fort Monmouth, New Jersey

INTRODUCTION

New memory systems are often developed as a result of the discovery and exploitation of physical phenomena which exhibit storage properties. As a result of such a "technique-oriented" approach, the structure and performance of the resulting memory system may be dictated by the intrinsic characteristics of the technique and it is possible that the most desirable storage characteristics, from the computer's viewpoint, may not be realized.

An alternative approach, which will be followed in this paper, is to establish a set of performance characteristics from a system's viewpoint and then to identify and examine a technique which can satisfy these guidelines. The objective of such a "system-oriented" approach is to develop memories which increase overall system storage efficiency by optimization of performance characteristics and by specialization of memory structure.

Specifically, the storage requirements of a data processing system are classified into basic functions with the type of information and its associated usage within the system forming the basis for differentiation. Rather than using a general purpose memory to perform all storage functions, a memory unit is designed for a particular function utilizing the specialized nature of the function as a basis for system optimization. Thus, the design procedure begins with an analysis of the class of data involved and uses the characteristics peculiar to this class of data to establish the performance requirements on a storage element and a memory system which will store this data most efficiently. Then, these performance characteristics are used to evaluate possible implementation techniques and if, as is often the case, current techniques fail to satisfy the technical guidelines established then these guidelines serve as an aid in the search for new techniques.

For example, the application of this "system" approach to memory design has resulted in the establishment of technical guidelines for block-oriented,[1] push-down list, and content addressable[2] memories which are associated with those block-oriented, list, and content addressable types of data and storage functions common to data processing systems.

Read-only memory functional requirements

Upon examination of the storage requirements of central processor memory, there is a "read-only" function and a class of "fixed" data which may be identified. The "read-only" function includes the storage of indirect accessing schemes, the implementation of logic functions, the storage of microprogrammed instructions, and related applications. The "fixed" data consist of known information which, by its nature, is relatively permanent and which must be available to the computer on a word basis. Standard subroutines, trigonometric tables, ballistic tables, language translators, and character generators are examples of this type of data and usage.

The primary characteristics of the memory evolve from (1) the specialized read-only function of the memory and the semipermanent nature of the data, and (2) the necessity for direct central processor control of the unit. Thus, the memory must provide nonvolatile storage with nondestructive read-out and the storage must be word-oriented with random access to the word. Following these basic definitions, additional characteristics may be derived.

Removable media

The flexibility of a read-only memory would be extremely limited if there were no provisions for altering the stored data. For example, in circumstances where different tables are required by a program, it

would be desirable to have a means of substituting this information without rebuilding the memory. This capability is provided if the memory has the data physically stored on a removable media.

However, associated with the concept of removable media is the problem of media registration. To provide adequate amounts of information for typical applications (without requiring the use of multiple read units), the capacity of a single media must be comparable to that of conventional (read/write) central processor memories. As bit packing densities are increased, correct alignment or registration of the media within the read unit becomes increasingly difficult. The difficulty lies in the coupling of the data on the media to the sense circuits of the read unit.

Tolerances on mechanical positioning of the media during and after insertion can become so critical that manual removal and substitution of the media is impractical. Schemes have been devised which attempt to alleviate the problem, but these schemes are still mechanically limited resulting usually in a restriction on the allowable capacity per media card. In many cases, it is this registration problem and not the storage capability of the media that dictates the overall capacity of the memory.

Read cycle time

Because the read-only memory performs a central processor function, it should provide access at speeds comparable to the primary central processor memory. Since the permanent nature of the storage eliminates the necessity for writing, rewriting, or erasing operations, it is reasonable to expect that the resulting memory organization should be extremely simple and that this simplicity should result in a significant reduction of access times as compared to conventional read-write memories.

Economic considerations

Due to its specialized function, its permanent storage, and the elimination of the write function, the memory should be less expensive than conventional memory types. If the cost per bit for both units is equal, then the particular application must be critically evaluated in terms of the remaining characteristics to determine if the use of a special purpose memory is justified. In fact, if any of the four distinguishing characteristics (i.e., removable media, faster access time, permanent storage, and lower cost per bit) cannot be achieved, then the storage function may be performed more economically, from an engineering viewpoint, by a conventional memory.

Proposed technique

Upon review of current techniques for implementation of the read-only storage concepts developed, the optical-photo methods appear most adaptable to fixed store. Photographic media are quite inexpensive, are capable of extremely high bit densities, and exhibit an inherent write-once, read-only storage capability. The optical read-out techniques, which are used, are nondestructive. On the other hand, such media are highly susceptible to scratches and dust particles; the accessing schemes are generally slow, requiring serial transfer of data; and the registration difficulties limit the overall bit capacity of the memory system below the capability of the media.

To overcome these problems and to retain the advantages of photographic emulsions, it is proposed to use holographic storage of binary information.[3] Holography, a new form of photographic recording, can introduce a unique type of redundant storage. For example, Leith[4] reports that diffused illumination holograms have an immunity to dust and scratches, and that particles have little effect in producing erroneous signals as in previous photographic memories. Also certain types of holograms act as a complete imaging system since they do not require lenses to project real images. This capability may indirectly ease registration tolerances and permit storage capabilities superior to other optical techniques.

It is intended that a hologram of a binary data array would constitute the card-like removable media. Upon insertion into the memory read unit, the hologram would continuously focus a real image of the data onto a photodetector matrix. Such an arrangement can permit electronic random access to the information within the array while eliminating the stringent optical requirements on the detectors involved.

Basic holographic principles

Holography, or imaging by wavefront reconstruction, is a two-step imaging process developed by Gabor.[5] The technique is based upon the illumination of objects by coherent light and the ability of such light to sustain a time invariant intensity distribution on a photographic plate. Although recent contributions have increased the complexity of the field, the underlying concepts remain the same.

General theory of holography

Construction: Figure 1 depicts the construction phase of the holographic process. Illumination of the target with temporally and spatially coherent light generates two fields at the photographic plane z_p. One field has not been altered by the object while

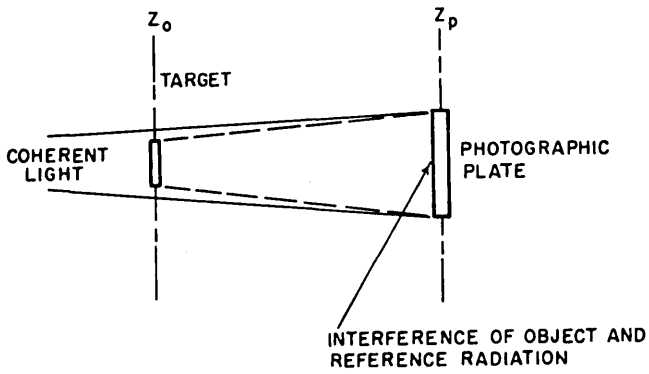


Figure 1 – Holographic construction

a second field results from object scattering. The total scalar field at the plate may be written as

$$E\exp(j\phi) = \exp(j\phi_r - j\omega t)[E_r + E_d\exp(j\phi_d - j\phi_r)] \quad (1)$$

where

$$E_d(x,y)\exp[j\phi_d(x,y) - j\omega t]$$
$$\text{and } E_r(x,y)\exp[j\phi_r(x,y) - j\omega t]$$

represent the modified and unmodified light respectively. The resultant power density (intensity) of the total field is equivalent to the square of the absolute value of the wave, or

$$I = [E\exp(j\phi)] \cdot [E\exp(j\phi)]^*$$

where * indicates the complex conjugate. Substituting Equation (1), one obtains

$$I = |E_d|^2 + |E_r|^2 + 2E_rE_d\cos(\phi_d - \phi_r). \quad (2)$$

Because photographic emulsions are sensitive to incident optical power, they will record this expression. Note that the time factor $\exp(j\omega t)$ has been eliminated, thereby making the power distribution fixed or stationary relative to time. Such a condition is maintained due to the coherent property of the light. Further, this expression exhibits information about the scattered phase ϕ_d as well as the scattered amplitude. The first and second terms of Equation (2) may be considered as D.C. components which contain little information but act as a bias level for the signal. The final term, representing the signal, expresses the modifications of light due to object scattering as the relative phase and amplitude variations with respect to the unmodified or reference light. The variations are functions of the spatial coordinates in the photographic plane but are invariant in time.

Through the ordinary photographic recording process, the energy distribution incident upon the plate during exposure is converted into a proportional value of real amplitude transmission t_n where t_n, in this case, is a function of Equation (2).

Reconstruction: The effect of this transmission can be seen in the reconstruction phase during which the negative is illuminated by a coherent beam, expressed as $E_c\exp(j\phi_c - j\omega t)$. The negative itself becomes an object scattering the incident illumination in proportion to the real amplitude transmission such that the transmitted field becomes

$$E_{trans} = E_c\exp(j\phi_c - j\omega t) \cdot t_n.$$

Substituting for t_n, its equivalent expression in terms of the original object field, one obtains a final value for the transmitted wave,

$$E_{trans} = k_nE_cE_r\exp(j\phi_c - j\omega t) \cdot [E_r + E_d\exp j(\phi_d j\phi_r)] + k_nE_cE_r\exp(j\phi_c - j\omega t) \cdot \frac{Ed^2}{E_r} + k_nE_cE_r\exp(j\phi_c - j\omega t) \cdot [E_d\exp(-j\phi_d + j\phi_r)].$$

where k_n is a constant of the photographic process. Comparing this equation with Equation (1), one can see the direct resemblance of the first term of the reconstructed wave to the original signal. It differs by a multiplier and a constant phase shift. The second term has little significance and may be considered as noise. The third term is critical, however, and is characteristic of the holographic process. It is generated by the modulating action of the emulsion and is therefore characteristic of all holograms. That is, because the emulsion acts essentially as a square law device, recording the product of the field and its complex conjugate, this final term will be produced in all holograms. It is frequently called the conjugate or twin image of the signal, for it contains the same amplitude as the signal but has opposite phase shifts relative to the background.

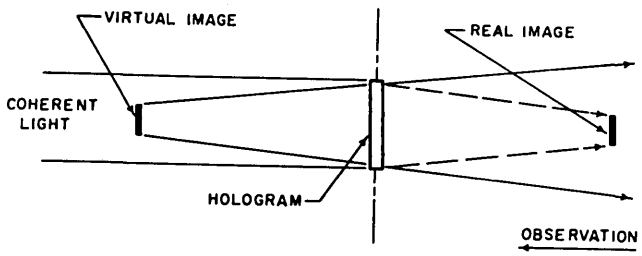


Figure 2 – Holographic reconstruction

When the hologram is viewed as in Figure 2, the signal term seems to diverge from an object behind the hologram plane, thereby appearing as a virtual

image. This signal is a replica of the field that would be produced if the object were in place. The conjugate term represents a real image whose rays converge to a plane in front of the hologram plate. When constructed in this manner, these real and virtual images are superimposed acting as mutual noise. The effect of such noise on the original signal is a major problem with Gabor's original technique, but such effects can be reduced by special construction methods, e.g., masking.[5]

The second term in Equation (3) acts as background noise for both the real and virtual images. Its effect may be reduced by controlling the reference light amplitude during the construction phase.[5]

Hologram storage characteristics

General discussion

The system characteristics for the memory have been established previously. The distinguishing memory characteristics of high speed, large capacity, permanent store, and low cost are the major factors in determining system feasibility. The additional factors of expandability, small size, low power consumption, and high environmental resistance are included to insure compatibility of the memory with the remainder of the computer system.

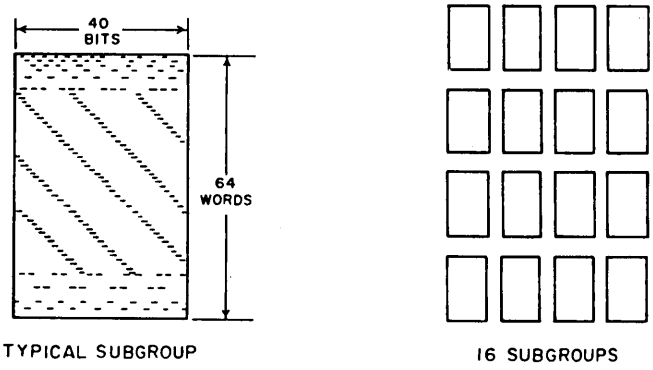


Figure 3 – Data array configuration

A diffused illumination, twin-beam hologram is to be constructed of a binary data array which consists of 1024 words of bit length 40. A possible configuration for the array may resemble Figure 3 which illustrates the placement of the data into sixteen subgroups each containing 64 computer words. The dimensions of the final array and bit configuration depend upon the capability of the holographic process and the physical limitations imposed by the read unit.

During reconstruction, the read unit would allow a real image of this array to be continually focused onto an optical read matrix, which consists of a photodetector for each bit location. Access may then be accomplished randomly by electronic selection and sensing of the individual detectors.

Information capacity

The digital information capacity of photographic emulsions has been shown to be extremely large.[6] Future photographic systems are predicted which have storage densities greater than those of contemporary storage techniques by a factor of 10^6, and this recording potential is surpassed only by genetic memories.[6] Analysis of emulsions on the basis of data capacity has been given extensive treatment in literature.[7] It has been demonstrated, to a first order approximation, that the information capacity in bits per unit area is equivalent to the square of the emulsions's resolving power. Numerically, then, Kodak 649-F plates (estimated to have a resolving capability of 2000 lines per millimeter) have an upper limit of 4×10^6 bits per square millimeter capacity.[7] Of course, attainment of such capacity depends upon the techniques of processing the data.

In terms of a one-to-one storage technique, a larger memory (4096 words) contains approximately 160,000 bits. If it is assumed that only 30% of 21-square inch media area is devoted to data, the bit packing density is only 40 bits per square millimeter. Obviously, this value imposes a modest demand upon the plate resolution.

Image Intensity: During reconstruction, it is desirable to transmit maximum light energy into the real image in order to maximize the electrical signal, improve signal-to-noise ratio, and compensate, to some degree, the variations in detector characteristics. With the holograms considered, there are several parameters that can be controlled to vary the real image intensity. The most obvious parameter is the output power of the reconstruction light source. It must be remembered, however, that since the outputs of present continuous wave lasers are proportional to the physical size of the device, there is a definite size limitation imposed by the memory on applicable lasers. Work on semiconductor lasers may prove useful in eliminating such a limitation. Secondly, it was noted that the image intensity is directly proportional to the amount of hologram area illuminated. Thus, a larger illuminating area seems advantageous. Thirdly, it may be shown by Equation (2),

$$I = |E_d|^2 + |E_r|^2 + 2E_rE_d\cos(\phi_d - \phi_r),$$

which relates the signal and bias levels, that a maximum signal is obtained by making the modulation

$\frac{2E_rE_d}{(E_d^2+E_r^2)}$ index, equal to unity. This may be done by controlling the object and reference beam levels with filters. With this index set at unity, the exposures may be controlled to permit operation of the photographic process in a linear region of the Hurter-Driffield[5] curve, thereby producing low plate densities after development. A low density plate transmits and, therefore, diffracts more of the incident light into the real image.

Resolution: Armstrong[8] demonstrates that the size of the hologram plate constitutes a finite aperture for Fresnel-hologram imaging systems. Mathematically this may be seen by extending the Fresnel process equation into two dimensions and integrating over the finite limits specified by the dimensions of the photographic plate. Upon performing the integration, it is found that the reduction in resolution is proportional to the constriction of aperture area. Thus, the introduction of particles or scratches decreases the resolution by reducing the effective aperture area below that of the plate. It may be stated that the resolution loss varies as the ratio of aperture sizes before and after the inclusion of a particle. This indicates that with half the hologram destroyed or blocked half the original resolution is obtainable.

Further, since the image intensity (assuming constant illumination) is dependent upon the illuminated area of the hologram, a reduction in the effective hologram area reduces the image power. This may be seen if one considers each hologram point as a lens which focuses into its own image. The total intensity of the image then is a function of the number of lenses or the illuminated area of the hologram. The amount of light diffracted into the image depends upon the source intensity and the area effecting the diffraction.

This variation of resolution and intensity as a function of plate aperture is not a characteristic of ordinary photographs. If half a photograph is destroyed, only half the image is visible, but the resolution and intensity of the remaining half has not been changed. On the other hand, if only a portion of a diffused illumination hologram is used, the entire image can be retrieved, but the intensity and resolution of this image is proportionally reduced. In digital storage applications, this effect may not be critical because half an image may be more necessary than half the resolution. The memory, for example, may tolerate reduction of resolution (assuming the initial value was adequate) but cannot tolerate the loss of any part of the image.

Registration effects

The coupling problem associated with most optical read memories may be divided into two major areas. First, there is the difficulty of converting the light signals into corresponding electrical outputs. This conversion becomes a function of the image intensity, and of the optical sensitivity and speed of the photodetectors. Second, there are problems in the registration, or alignment, of the media and the light signals with respect to the detectors. These problems are related to the bit size and the mechanical limitations imposed by the removable media requirement. The registration problem is not peculiar to photographic memories but stems from the high bit density and the removable media criteria. The difficulty can generally be traced to the storage scheme which, for most memories, is on a one-to-one basis. Each data bit, then, is allocated a physical position in the memory. The media placement must register each bit location onto a corresponding sensor location.

To briefly demonstrate the superiority of hologram registration, consider a possible random-access photographic memory which uses bit-by-bit storage. The media is a photographic picture of a data array which consists of appropriate transparent and opaque areas. This array is focused by a lens system onto an associated photosensor matrix. The positioning of the media with respect to the lens system must permit focusing of a single bit such that the movement or displacement of the media does not result in a movement or displacement of the image more than a fraction of the bit size. Therefore, the mechanical holders for the media must maintain placement to within a bit size, assuming a direct relationship exists between media and image positions. Numerically, assume that the memory specifications require 1024 words of 40 bits each to be stored on a 21-square inch media. This demands a maximum bit length of .0224 inch. Positioning accuracies should be well within one quarter of this size, allowing a .0056-inch positioning tolerance of the media in the media plane. This one-quarter derating factor must reflect the resolution available, diffraction effects, lens distortion, and photodetector characteristics. It may be that this value is too large and greater mechanical accuracies are necessary.

Now consider the holographic reconstruction process depicted in Figure 4. A diffused, twin-beam, Fresnel hologram is used to generate a magnified real image at a finite distance from the plate. A normal incident beam will, in the case illustrated, produce a diffracted, first-order, magnified image. It is evident that a transverse shift of the hologram by a distance d across the beam will cause the real image to move the same lateral distance d. Therefore, even with magnification, there is a unity factor between the positioning of the media and the image.

Although the Fresnel holograms seem to offer an improvement over conventional photographic tech-

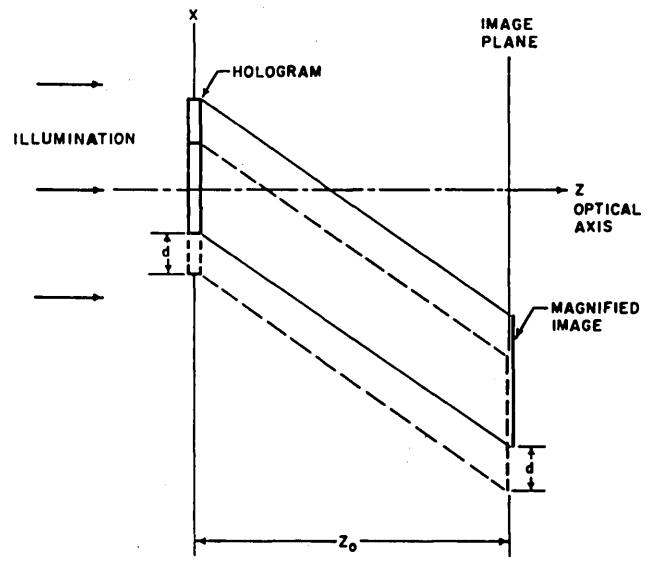


Figure 4 – Lateral shift of plate, Fresnel hologram

niques, an attempt was made to determine a more optimum condition for image registration. In this respect, Fraunhofer holograms appear to offer significant advantages. Figure 5 shows the Fraunhofer construction of an object composed of an infinite number of line sources. Because each source is placed at the

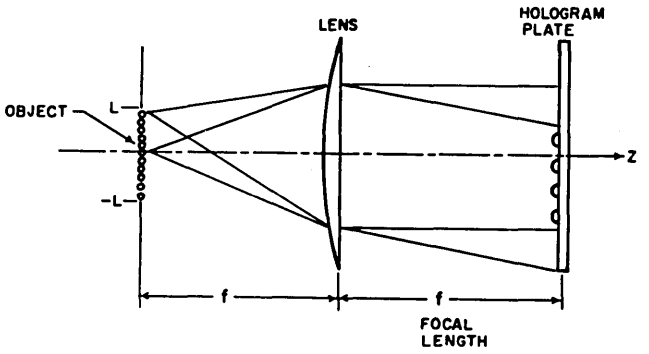


Figure 5 – Construction of object at infinity

focal plane of the lens, it will generate, in conjunction with the lens, a plane wave emanating at a predictable angle. Each source then creates a corresponding plane wave across the plate. The addition of a reference beam produces a sinusoidal intensity distribution on the plate for each source. Interference between source components is not considered. Because the distribution for a single source is similar over the entire plate in one linear dimension, illumination of any section during reconstruction will produce the same result. That is, employing a reconstruction beam whose illuminating area is less than the plate area causes plane waves to be emitted and, therefore, an image to be produced whose angular position does not depend on the hologram's linear displacement in one dimension, but which is a function of the incident beam position. As the hologram is moved across the beam, the angular position of the image does not vary (cf Figure 6). However, unlike the Fresnel type of hologram, a lens is required to produce a reconstructed image at a finite distance. Figures 7 and 8 illustrate the characteristics of diffused illumination Fraunhofer reconstructions with respect to image registration. Although the preceding discussion does not consider all of the possible displacements of the hologram and the resulting effects on the reconstructed image, it does serve to illustrate some of the more significant advantages of the technique.

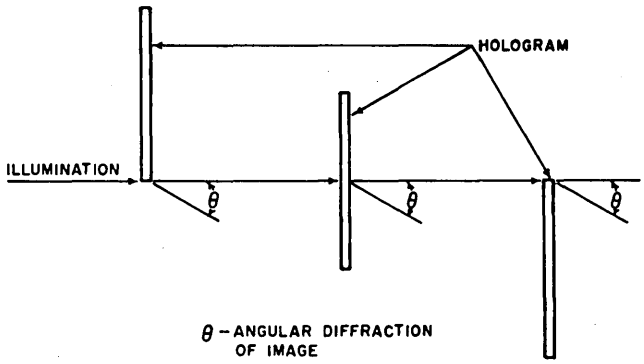


Figure 6 – Reconstruction with image at infinity

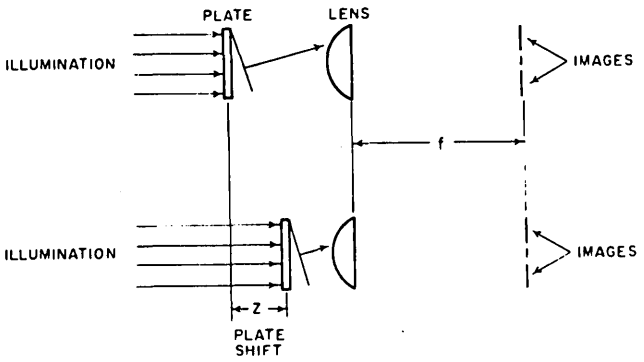


Figure 7 – Z axis displacement, Fraunhofer hologram

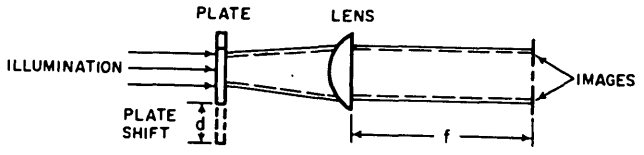


Figure 8 – X or Y axis displacement, Fraunhofer hologram

System design considerations

As previously stated, one approach to the implementation of a removable media, read-only memory by holographic techniques is as follows. The removable media would consist of a hologram of a binary data array. The actual dimensions of the array and the final configuration of the data will depend upon the capability of the holographic process and the physical limitations imposed by the read unit. During reconstruction, the read unit would allow a real image of the array to be continually focused onto an optical read matrix consisting of a photodetector for each bit location. Access may then be accomplished randomly by electronic selection and sensing of the individual detectors. It is proposed to use a laser to provide the coherent illumination.

In the design synthesis, advantage should be taken of the specialized nature of the read only function. For example, effort should be made to decrease access time and to implement the system with a minimal amount of hardware. In this case, the effort will consist of gaining simplicity in the control block and in the remaining circuitry peripheral to the detector matrix. Overall, the memory must provide a word-oriented, random access, nondestructive read capability while being compatible with the remainder of the computer through the processor-memory interface. Figure 9 shows a block diagram of the proposed memory design.

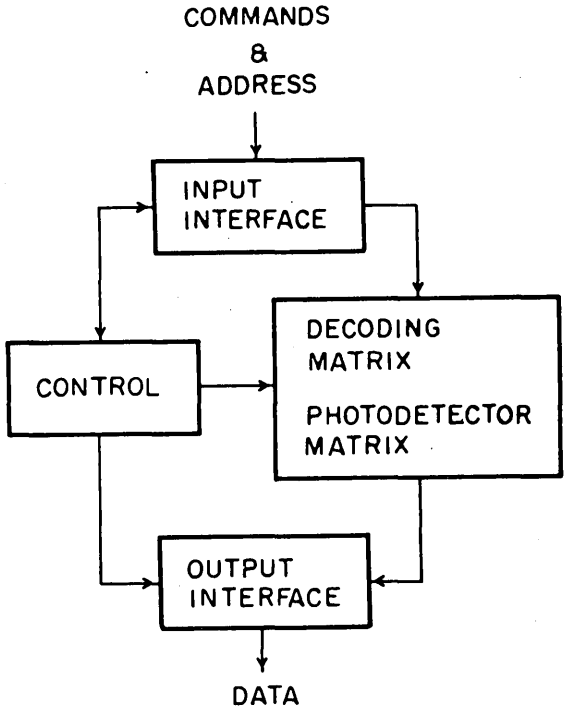


Figure 9—System organization

Read matrix requirements

The read unit matrix, in conjunction with the control section, must accept the address, decode the address, and place the data into the sense amplifiers. The matrix consists of a 40,960-bit planar array of detectors and associated address-decoding circuitry. Figure 10 shows a simplified matrix which demonstrates the access technique. Here, a diode decoding matrix, driven by the address line drivers, $A,\overline{A},B,\overline{B}$, exhibits a word-oriented, random access, linear select function. The speed of decoding depends upon the diode switching speeds which, in turn, are functions of diode capacitance. Extension of this simple matrix to the full tance. Extension of this simple matrix to the full 1024-word memory would indicate, due to the number of diodes involved, the necessity of address drivers to insure a .1-us access time. Such an expansion would also indicate the 1024 lines connecting the decoding circuitry to the photodetector array and the need for completing these connections during fabrication of the matrix. It may be possible to utilize the photodetectors in the array for address decoding if the proper addressing is included in the image of the hologram.

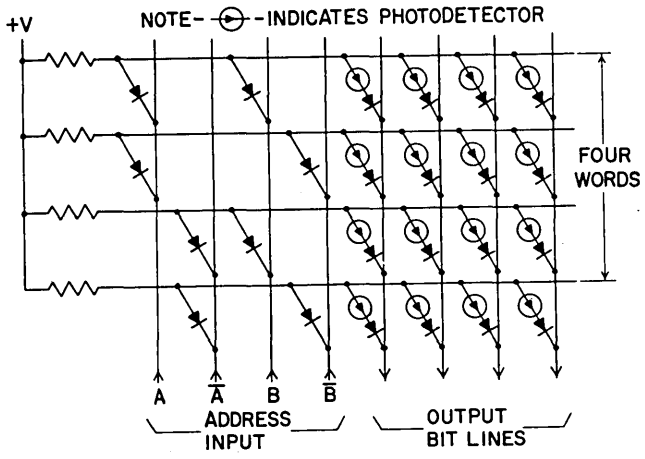


Figure 10—Simplified read matrix

Each bit location, or cell, contains an electrically fast, highly sensitive photodetector and a high speed isolation diode to prevent crosstalk. The electrical characteristics of both the photodetector and the isolation diode affect the read cycle. To increase the speed, the capacitance per cell should be reduced to the lowest possible value. It is conceivable to use a driver per word line or a driver per element although such additions would increase matrix complexity beyond present day practical realization.

Detector Study: It should be apparent that many of the matrix characteristics depend upon the type photodetector to be used, and, indeed, the photodetector

properties may dictate system feasibility. The principal concern is the choice of appropriate detectors and, from those available, which can be fabricated by simple techniques. In choosing the most suitable detector, the main properties that are required are optical sensitivity, electrical response, and ease of fabrication. The sensitivity of each detector must be high due to the relatively low optical power usually concentrated in the hologram real image.

In examining the electrical speed of the detectors, one must differentiate between optical and electrical switching characteristics. To illustrate these differences, consider a photodetector which is maintained at the proper electrical bias. If it is exposed instantaneously to a level of light, the optical response or switching time will be defined as the time required to electrically detect (at the output terminals) the total light change. This time includes the delay through the device and the rise time of the electrical variable, voltage or current. If, on the other hand, light is constantly on the detector, and the electrical bias is applied as a unit step function, the time required to fully sense the state of the detector will be referred to as the electrical switching time.

A survey of several optical detectors, including a discussion of the basic physics of their operation, can be found in the literature.[9,10] The devices which are applicable, especially from the standpoint of fabrication, are photoconductors, photodiodes, and phototransistors. In terms of the distinguishing characteristics dictated by the memory, i.e., sensitivity, speed, and ease of fabrication, all three detector types seem applicable. Because the image of the data array will be continually focused onto the detectors, the electrical switching characteristics are most interesting. From this standpoint, perfect photoconductors act simply as resistors, and therefore the electrical switching response is dependent upon its conductance and the impedances of the pre- and post-circuits. With good circuit design, this response becomes that of the pre- and post-circuits which, in this case, includes the decoding circuitry for the matrix. (Practically, there will be some parasitic capacitance associated with each detector element.)

Experimental results

A number of basic experiments were performed in an attempt to verify the predicted characteristics of holograms with respect to the read-only memory application. The primary areas of interest were the image intensity, image resolution, and registration effects associated with the construction and reconstruction of holograms storage images of two dimensional digital arrays.

The experiments confirmed the predicted registration effects and the application of holographic techniques to the read-only memory problem appears to offer significant advantages in the area of image registration with respect to the detectors. Measurement of the image resolution indicated that the resolution achieved was in the order of 13 lines/mm. However, due to the "granularity" which appears under visual observation of the reconstructed image, it is felt that the measurement was limited by the technique and that the actual resolution exceeds this figure. In addition, the resolution is also a function of the sophistication of the experimental facilities and technique and this value should not be interpreted as a maximum limit. Image intensity requirements are obviously dictated by the sensitivity of the selected photodetectors. To obtain some indication of the adequacy of image intensity for the intended application, as well as the overall feasibility of this approach, a breadboard model was constructed to actually read digital information from a hologram.

The breadboard model as shown in Figure 11, consisted of (1) a main chassis containing address selection switches, sense amplifiers, a four-bit output register and associated indicators, and (2) an auxiliary chassis containing a 4 × 4 array of photoresistors with an associated diode selection matrix. The photodetectors were organized into four words of four-bits each and were located in the auxiliary chassis to facilitate positioning with respect to the reconstructed image. A Neon-Argon gas laser with a power output of 0.6 mW (fundamental mode) was used as a source to reconstruct a hologram of a 16-bit data array. When the photodetector array was illuminated by the reconstructed image of the data array, it was possible to select and read out a four-bit word.

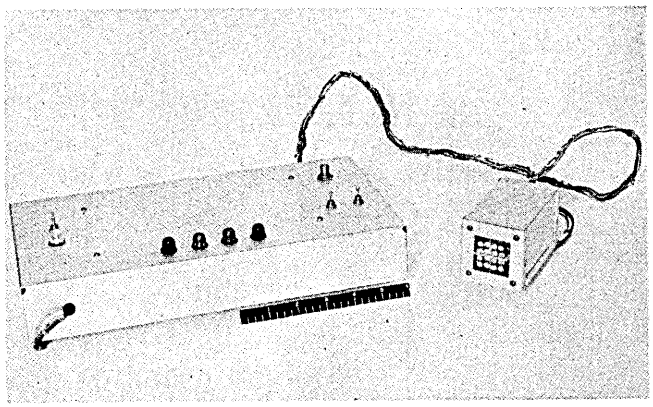


Figure 11 – 16-bit breadboard

Alternate approaches

The proposed approach for implementing a read-only memory using holographic techniques specifies a single reconstruction source and a single photodetector array with sufficient capacity to read the contents of a 1024-word memory. However, the ultimate feasibility of the technique need not depend on the development of such a photodetector array. Alternate configurations are possible which would allow the use of detector arrays with fewer photodetectors without reducing the total storage capacity of the memory.

For example, by using two reconstruction sources and by constructing two different holograms on a single photographic plate (each covering one-half the plate), the number of photodetectors in the array may be reduced by one-half. Each hologram would be associated with one of the reconstruction sources and would be constructed to cause proper imaging on the photodetector array upon illumination by the associated source. Then, access to a desired word would involve the selection and triggering of the proper source in addition to the normal selection in the photodetector array.

Of course, this scheme would require that the optical response of the photodetectors be less than 100 ns since the illumination of the elements in the array is no longer steady-state. This requirement would indicate the use of photodiodes or phototransistors in the photodetector array.

CONCLUSIONS

In general, the results of this study have indicated that holographic techniques are particularly suited to satisfy the functional requirements of read-only memory as specified. For example, holography offers solutions to two key problems associated with the requirement for a single removable media storing up to 160,000 bits. First, the unique redundance inherent in holograms constructed with diffused illumination eliminates the loss of data due to such environmental effects as dust and scratches. Second, the potential freedom from registration effects which can be achieved by proper selection of construction techniques allows the manual insertion and removal of media with high bit packing densities and does not add a requirement for complicated mechanical positioning or complex electrical interconnection in the read unit.

Although the initial construction of a storage media by holographic techniques requires a coherent light source and a fairly stable system, these requirements are within the scope of the system functional requirements, as previously defined. Moreover, the requirements for the reconstruction or reproduction of a storage media are much less stringent and appear to be capable of allowing the development of a read unit which is suitable for operation in a field environment. It has been shown that holograms can be reproduced by contact printing and that holograms can be reconstructed by noncoherent light sources.

Of course, the development of a feasible holographic read-only memory does not depend solely on holography. It is apparent that the ultimate success of this concept also requires the existence of photodetector matrices with sufficient size, sensitivity, and speed for this application, and which, in addition, are capable of economic fabrication. The development of photodetector matrices with the required characteristics appears to be within the state-of-the-art and should present no insurmountable obstacles. Indeed, if a "multiple-source/multiple-hologram" approach is selected, the number of bits required for the photodetector matrix may be small enough to allow practical and economic fabrication of the matrix on a discrete element basis.

REFERENCES

1 B H GRAY D R HADDEN JR. D HARATV
Block oriented random access memory
Paper presented at the IEEE Conference on The Impact of Batch Fabrication on Computers Los Angeles Calif 1965

2 A V CAMPI R M DUNN B H GRAY
Content addressable memory system concepts
IEEE Transactions on Aerospace and Electronic Systems Vol AES-1 No 2 1965

3 D R HADDEN JR.
Private communications, Comm/ADP Laboratory, USAECOM Ft Mon N J 1965

4 E LEITH J UPATNIEKS
Wavefront reconstruction with diffused illumination and three dimensional objects
J Opt Soc Am Vol 54 p 1295 1964

5 D GABOR
Microscopy by reconstructed wavefronts
Proc Rov Soc London Vol A197 p 454 1949

6 M CAMRAS
Information storage density
IEEE Spectrum Vol 2 p 98 1965

7 C McCAMY
On the information in a microphotograph
App Opt Vol 4 p 405 1965

8 J ARMSTRONG
Fresnel holograms: their imaging properties and aberrations
IBM Jour R&D Vol 9 p 171 1965

9 D CADDES B McMURTRY
Evaluating light demodulators
Electr Vol 37 p 54 1964

10 R KAUS
1965 survey of commercial semiconductor photosensitive devices
Electr Ind Vol 24 p 82 1965

Low power computer memory system

by D. E. BREWER
Air Force Avionics Laboratory
Wright-Patterson Air Force Base, Ohio

and

S. NISSIM and G. V. PODRAZA
The Bunker-Ramo Corporation
Canoga Park, California

INTRODUCTION

One of the critical technologies for aerospace missions is electrical power. Increased emphasis is being given to reducing power requirements of electronics as well as toward improvements in power sources themselves. In present day aerospace computers the memory subsystems consume approximately 40 to 60% of the total computer power. This is primarily due to the high drive requirements of present core memories or to the sensing problems associated with thin film memories. Reduction in memory power requirements would, therefore, have a significant effect on extending allowable mission time.

The objective of the effort described in this paper is to demonstrate a capability for the design and construction of an aerospace computer memory having very low power requirements. The device technology selected to accomplish this objective is the metal oxide silicon (MOS) transistor memory array. The MOS transistor is an insulated-gate field effect device formed in a silicon crystal and is suited for low-power applications. The idea of using active devices for storage of data in computers is as old as computer systems. However, until very recently, the power required and the cost of active memories were prohibitive for almost all applications. With the progression of integrated circuits technology from the gate functional level to very large scale integration of system functions, considerable interest has been devoted to development of monolithic memories.[1-4] Predicted power requirement for a random access memory subsystem using monolithics varied over a wide range depending on various proposed designs. This effort primarily emphasizes low power circuit design and memory organization concepts leading to a low power aerospace computer memory that is suitable for economical production.

As the feasibility vehicle for this approach, a 1024-word, 30-bits/word, random access memory was constructed using P channel MOS transistor arrays for storage and bipolar transistors for the peripheral accessing circuitry. This system is an NDRO, electrically alterable, memory with a read access time of 0.7 microseconds, a read or write cycle time of 1 microsecond and has a total power requirement of approximately 3.5 watts. This unit has been successfully constructed and delivered to the Air Force Avionics Laboratory. Significant results were obtained in the areas of low-power circuit design, memory design, and organization concepts using MOS transistor arrays. All of the areas will be covered in detail in this paper.

Outline of technological approach

review of the circuit operation if the basic memory cell

The MOS integrated circuit approach was chosen as a means of developing a low power memory array for two reasons, namely (1) MOS bistable arrays can be operated with very low power by using pulsed operation techniques, and (2) this approach is well suited to large scale integration and reasonable yield, an essential prerequisite from an economy viewpoint.

The peripheral circuitry, which included the address decoder, write drivers, sense amplifier, and control circuits, utilized bipolar transistors and employed commercially available integrated circuits whenever possible. The MOS-to-peripheral system interface circuits were fabricated from discrete components because voltage levels involved in these circuits were not compatible with the relatively low voltages of bipolar inte-

grated circuits. A savings in power is achieved in the peripheral circuits by employing techniques which minimize power until circuits are actuated by pulse signals. Further power reductions were realized by employing special nonlinear switching circuits with inductive source impedances to decrease the power consumption when driving large capacitive loads in the MOS array and distribution lines.

The system development included a memory exerciser to aid in testing and in demonstrating operation of the memory, and a power supply operated from a 110 volt, 60Hz source to furnish the +13, +12, +4, −4 and −7 supply voltages required by the memory. The exerciser and power supply are of conventional design, were not concerned with power conservation, and therefore, will not be described in detail.

Description of the MOS memory chip

The memory cell circuit, monolithic configuration for the 64-cell chip, and basic principle of low power operation are described in a companion paper.[5] For the sake of completeness in the present paper, a brief review of the circuit operation of the basic memory cell is presented in this section.

Figure 1 is the schematic of the memory cell utilizing P channel enhancement mode, MOS integrated transistor structures, having a nominal threshold voltage of 5 volts The substrate is operated at a 12 volt potential. Transistors Q_1 and Q_2 form the basic bistable storage element of the cell. Transistor Q_3 serves as a "load" for Q_1, while Q_4 is the load for Q_2. Transistors Q_5 and Q_6 are biased off during quiescent memory operation (i.e., neither reading nor writing).

Assuming Q_1 in the bi-stable is on and Q_2 off, the node voltage at the drain of Q_1 will be +12 volts, while the Q_2 node will be near ground potential. During quiescent operation, the capacitance of the Q_2 node will begin to charge toward +12 volts because of the leakage current through the P-to-substrate junctions common to this node. These junctions are constituted by the drains of Q_2 and Q_6, the source of Q_4, and any interconnection crossunders that utilize a P-diffusion onto the substrate. The voltage buildup on the node capacitance by leakage would eventually cause Q_2 to turn on and cause possible loss of the logic state of the bistable. To prevent this, the restore pulse is periodically applied to Q_3 and Q_4, resulting in discharge of the Q_2 node. The Q_1 node voltage is not appreciably affected by restore, since Q_1 is biased on and has a much greater transconductance than Q_3. Periodicity of restore must be sufficient to keep the Q_2 node properly discharged under condition of worst-case leakage current. The restore pulse for each cell

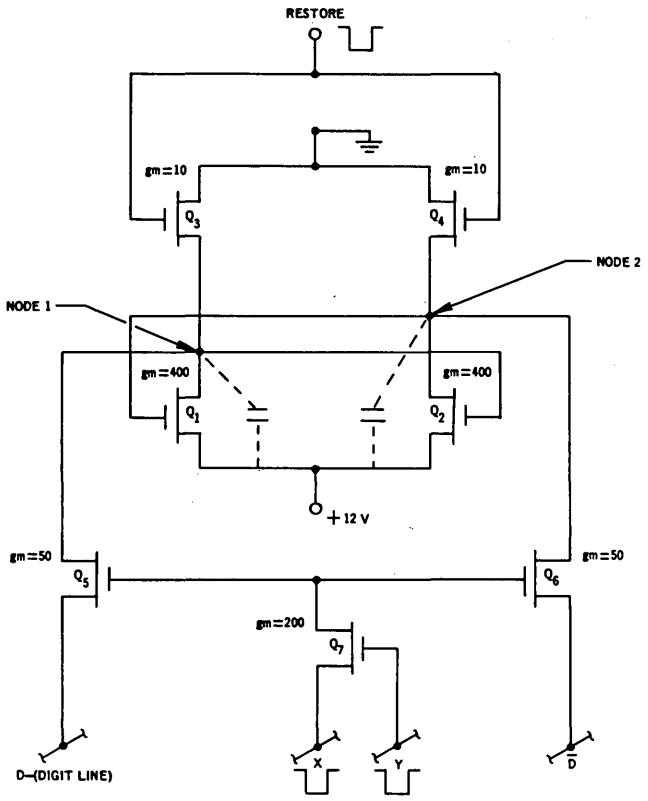


Figure 1—Schematic for basic memory cell circuit

is 18 volts in amplitude, 1.5 microseconds in width and has a repetition rate of approximately 10 KHz.

When reading from the cell, or writing information into the cell, use is made of the Q_5 and Q_6 gating transistors. The cell is addressed by applying the X- and Y-address pulse, as shown in Figure 2 and 3, to the respective source and drain electrodes of Q_7. Coincidence of these address pulses causes Q_7 to conduct, biasing the gates of Q_5 and Q_6 on. The Y-address pulse must have a longer duration than the X-pulse in order that Q_5 and Q_6 be turned off by the transition of the X-address pulse back to +12 volts.

Writing is accomplished by applying a write pulse (Figure 2) to the cell simultaneously with the address pulses. With Q_1 assumed conducting, and Q_2 off, a logic "1" is stored in the cell. If a "O" is to be written into the cell, the write pulse, which rises from ground to +12 volts, is applied to the digit complement while the digit line is held at ground potential.. This causes Q_6 to conduct current into the capacitance at the Q_2 node, resulting in turn-off of Q_1. The Q_1 node is now discharged toward ground through Q5, resulting in the turn-on of Q_2. With termination of the address and write pulse signals, the bistable is then retained in the zero state.

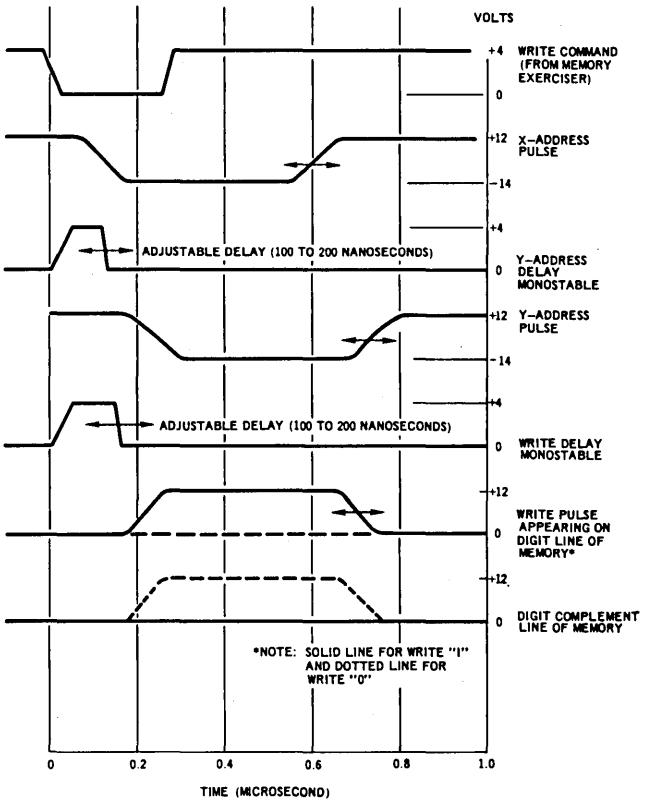


Figure 2—Memory timing diagram for write cycle

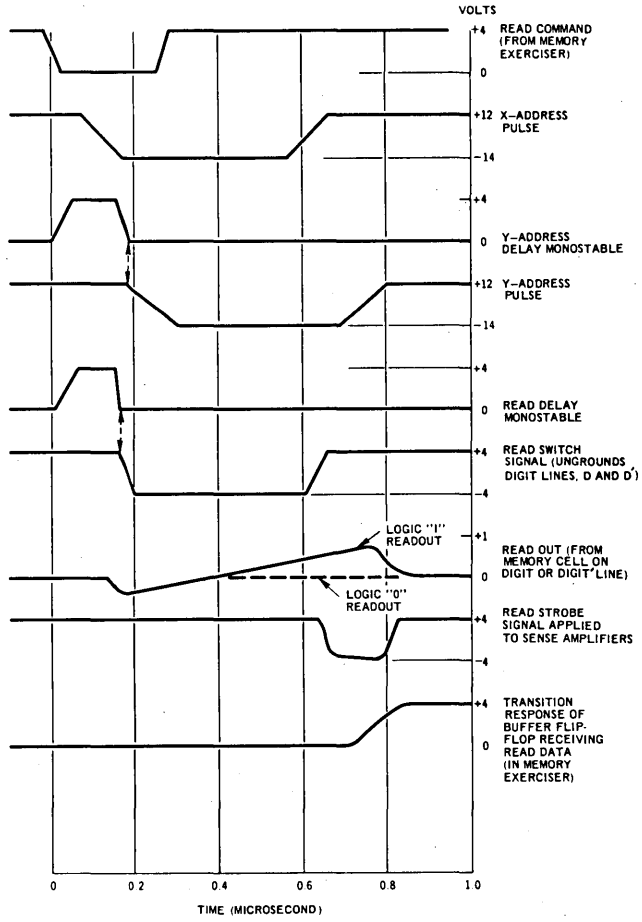


Figure 3—Memory timing diagram for read cycle

Readout of the cell (Figure 3) is accomplished by addressing the cell in the same manner as for writing. Both the digit and digit complement lines are near ground potential for read. When addressed, current from the "on" transistor in the bistable will flow into the respective digit or digit complement line. The node of the "off" transistor will be near ground potential ahd, hence, will not cause a current flow into the corresponding digit line. A differential amplifier, connected to the D and $\overline{D}$ digit lines, senses the readout.

The monolithic chip, shown in the microphotograph of Figure 4, measures 80 mils by 100 mils and contains 64 memory cells interconnected to form one bit of 64 different words. One memory cell occupies an area of approximately 100 mils^2 on the chip. The chips were individually housed in 22-lead flatpacks. The monolithic parameters of importance, external to the flatpack, are the capacitances looking into the restore, the digit and the address terminals. These capacitance values are shown in Table I based on averages of several flatpack samples.

Table I

Capacitance to Substrate Measured on 64-cell Monoliths Housed in 22-Lead Flatpacks

Measured Point (to substrate)	Typical Capacitance (picofarads)
D line	17.08
D′ line	16.75
X line	7.00
Y line	7.48
R	26.5

System description

Figure 5 shows the basic organization of the system. The memory stack is comprised of the MOS transistor arrays described in the preceding section, while the peripheral circuitry is composed of the address decoder, write drivers, sense amplifiers and timing circuits. In the case of this feasibility model, the memory

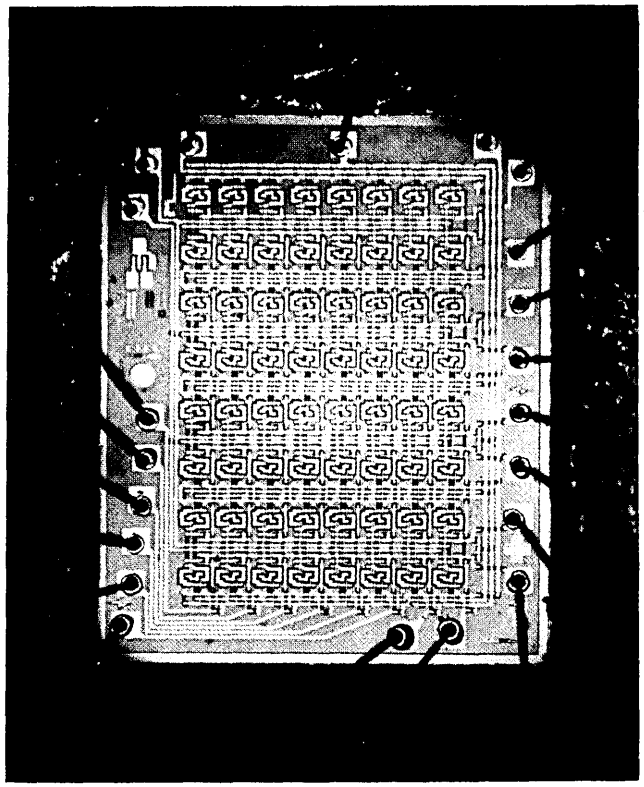


Figure 4—Monolith arrangement of 64 memory cell circuits

exerciser system (not shown) was built into the system to serve as the data processor. The exerciser supplies write information, and accepts read information from the memory, and also has the capability of detecting and locating any digital errors created in the memory system in terms of the word address and specific bits in the word.

As shown in Figure 5, the 10-bit address input is divided into a 5-bit section for decoding in 32 X-lines and a similar 5-bit section for the Y lines. A common pair of digit lines (Data and $\overline{\text{Data}}$) connects all 1024 memory cells corresponding to a specified bit in each word. These digit lines are multiplexed to carry both the write and read information to and from the memory.

Each 64-cell chip in Figure 5 is labeled with the address line designations which it receives. The columns represent the bit or digit positions, while each row represents a 64-word group.

Bipolar transistors were found advantageous for driving the large capacitances associated with the memory arrays. These bipolar peripheral circuits are individually described in the following paragraphs.

Address decoder

The 20 address input lines (10 digits plus complements) supplied from an address register (contained in he exerciser unit) represent a capability for addressing the 1024 words in the memory. Decoding is achieved in a conventional manner for a two coordinate addressing system. Figure 6 presents a block diagram of one of the two identical X or Y decoders. Three of the five digit inputs are decoded by eight NAND gates, labeled 0 through 7, while the remaining two digits are decoded by four NAND gates (A through D). These respective NAND gates drive pulse activated transformer circuits which in turn drive a 4 $\times$ 8 matrix of transformer circuits for the 32 output lines.

Schematics associated with the various blocks are shown by inserts in Figure 6. Commercial integrated circuits were employed for the NAND gates. One high level matrix driver and one low level matrix driver is actuated in correspondence with the address code when the address command pulse occurs. The two actuated circuits in turn select one of the 32 identical address driver circuits in the address matrix corresponding to the circuit path from the high level driver through the transformer winding to the ground return provided by the low level driver. Operation of the address driver circuit is discussed in a subsequent section dealing with power conservation.

Write drivers

Sixty input data lines (30 digits plus complements) representing the information to be written into the memory, are supplied from the memory exerciser. Each input line is supplied to a write driver circuit. Upon receipt of the write command, the lines having a logic "1" state activate the respective write driver for either the D or D digit lines.

The schematic for the write driver is shown in Figure 7. The digit line associated with each write driver is grounded through R_8 and Q4 during quiescent operation. When writing, Q_2 is turned on if the transformer is energized, while Q_3 and Q_4 are driven off thus allowing the twelve volt write pulse to be developed on the digit line. When the pulse terminates, Q_2 is driven off by the reversal of voltage across the secondary windings due to "flyback" action[6] of the transformer, while Q_3 and Q_4 are driven on. This rapidly discharges the digit line capacitanct and clamps the digit line back to ground.

Read sense amplifiers

Similar to the write drivers, each of the 30 sense amplifiers connects to the main digit (and digit complement) line which ties together all the individual digit lines from the 16 flatpacks corresponding to a

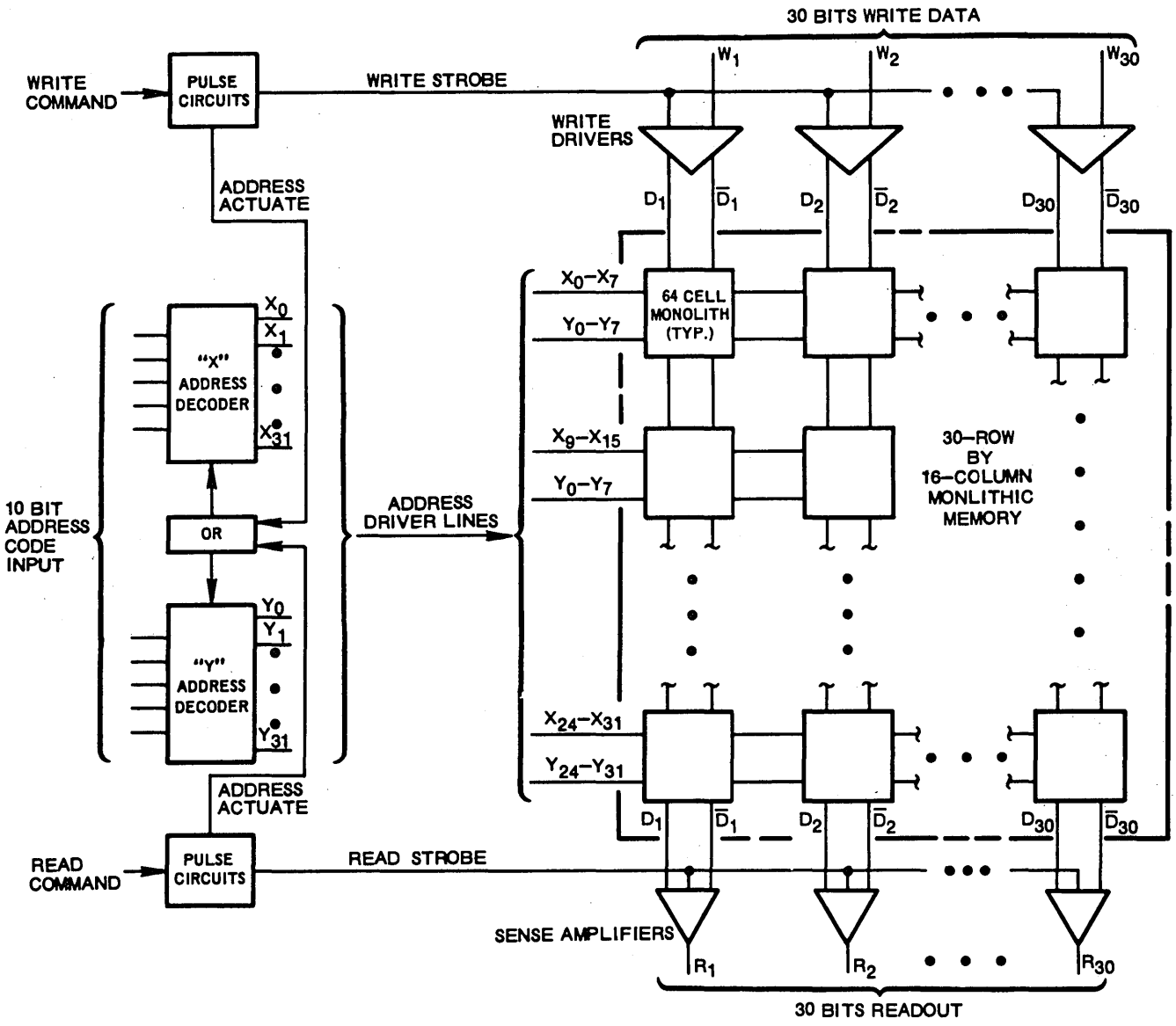


Figure 5—Block diagram of memory system

specified bit position. When a read command is received, the memory cells of the addressed word supply current to the digit lines. The differential between the digit and complement line signals are sensed to produce the corresponding read output when the strobe pulse is applied by the read strobe generator circuit.

The real sense amplifier schematic is shown in Figure 8. In consists of a differential amplifier (Q_5 and Q_6) which "primes" a flip-flop comprised of transistors Q_1 and Q_2. When the negative strobe signal is applied to the flip-flop it is actuated to the state determined by the differential amplifier. The sensitivity of the sense amplifier is improved by isolating the flip-flops from a possible unbalanced output loading by employing buffer transistors Q_3 and Q_4.

Restore circuits

The restore pulse, as previously described, is applied periodically to each cell in the memory to discharge the voltage buildup, due to leakage current, on the intrinsic capacitance at the flip-flop node.

To prevent large surges of current that would be required if the restore signal were applied to the entire memory simultaneously, the memory is divided into eight sections. Figure 9, showing the restore circuitry, consists of a 3-stage counter and associated binary-

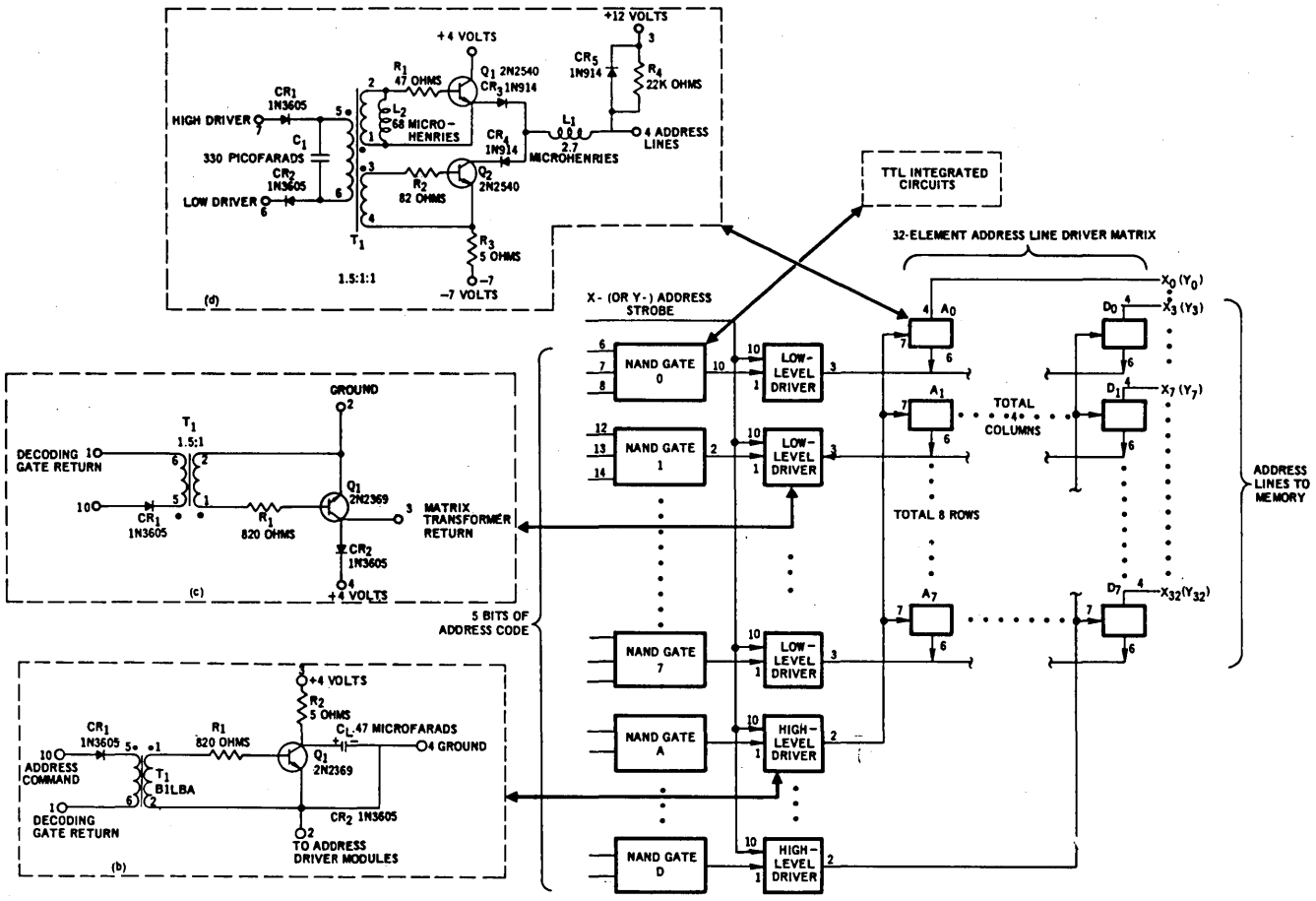


Figure 6—Address decoder block diagram with circuit schematics

to-octal decoder, with each output of the decoder controlling a respective restore pulse generator. A self-contained restore clock, asynchronous with the rest of the system, generates the shift pulses for the counter. Each time a restore clock pulse is generated, the pulse gate corresponding to the actuated line of the octal decoder delivers a pulse to its respective restore driver. The restore driver schematic is shown in the insert of Figure 9.

Timing control circuits

The control circuits cause the memory to be addressed whenever a read or a write command is received, and they generate the corresponding activating signals for the sense amplifier or the write driver circuits. Figure 10 shows a block diagram of the control circuits, with integrated monostables used for timing control. The schematic for the address command pulse driver, shown in Figure 11 is typical of the output driver circuits in Figure 10. Signal timing diagrams for the write and read cycles, respectively, are shown in Figures 2 and 3.

Physical arrangement

The physical packaging arrangement utilizes four multilayer printed circuit boards — two for the memory stack and two for the peripheral circuitry. The X-address decoder and the restore generator are packaged on one of the periphral circuit boards. The Y-address decoder and the control pulse circuitry are packaged on the second peripheral circuit board. No attempt was made to achieve a high degree of miniturization in this developmental effort.

The memory array consists of 480 identical flat-packs arranged in a 16 by 30 matrix and packaged on two multilayer printed circuit boards. One of these boards with the full complement of 120 flat-packs is shown in Figure 12.

The memory, exerciser, and power supplies are packaged in a Samsonite suitcase with exterior dimen-

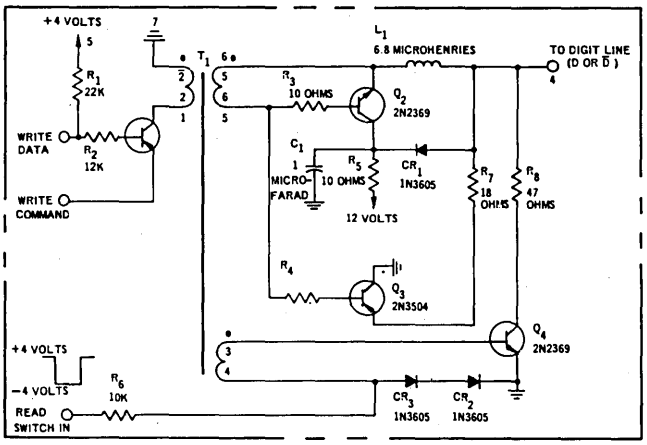


Figure 7—Write driver schematic

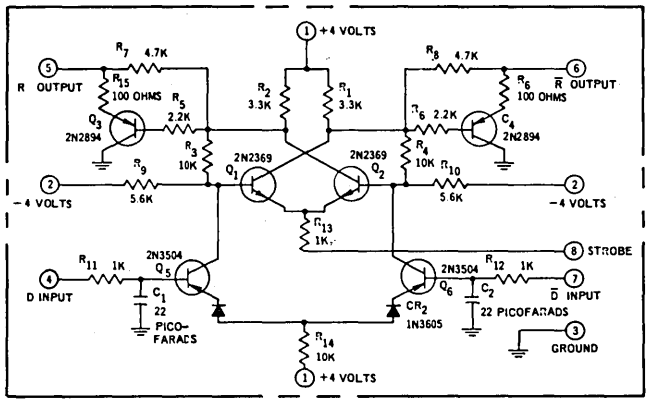


Figure 8—Read sense amplifier schematic

sions of 6 by 15 by 21 inches. The objective of this packaging configuration was to make the feasibility model of the MOS memory completely portable for demonstration purposes.

Power reduction techniques

The primary means of conserving power in the MOS memory cells was the use of the previously discussed restore technique. The power required per memory cell then becomes a function of the restore duty cycle. Using a 10 KHz restore rate resulted in a quiescent dissipation of 30.3 microwatts per cell, thus the 30,720 cells for the memory consume 930 milliwatts.

Table II

Capacitance Loads Presented to Peripheral Drive Circuitry

Circuit	Measured capacitive component (picofarads)
X-address line (120 flatpacks)	1,320
Y-address line (120 flatpacks)	1,170
Digit line, D (16 flatpacks)	349
Digit complement line, D′ (16 flatpacks)	380
Restore line (60 flatpacks)	3,500

Relatively large capacitance loading is presented to the address, write driver and restore cricuits by the memory as shown in Table II. The power consumed in charging and discharging the line capacitance using a resistance driver source is given by

$$P = CV^2 \times (p.r.f.)$$

where C is the effective capacitance of the memory line, V is the peak pulse voltage and p.r.f. is the pulse repetition frequency at which the capacitance is charged and discharged. For the 1200-picofarad address line capacitance and a 26 volt address pulse, each of the two address drivers would dissipate 0.8 watts, requiring 1.6 watts for address line driving if a resistive source impedance had been used in the line driver design.

The method used for reducing power dissipation is to make the line driver have an inductive source impedance, so that a resonant circuit results, and develops the desired voltage amplitude on the line capacitance in conjunction with nonlinear switching techniques which preverts "ringing" by the resonant circuit. The principle involved is illustrated by the simplified schematic and waveform sketch in Figure 13, where ideal components are assumed. The capacitor in Figure 13a is assumed initially charged to $V_c(O)$. When S_1 is closed at t = O, the voltage across the capacitor is given by:

$$V_o = V_1 + [V_c(O) - V_1] \cos \omega_o t$$

where: $\omega_o = \dfrac{1}{\sqrt{L\,C}}$ and

V_o = output voltage across the capacitor

When $t = \dfrac{\pi}{\omega_o}$, the voltage, V_o, in the equation would reach a peak value of $2V_1$—$V_c(O)$. For example, if $V_1 = -7V$ and $V_c(O) = 12V$, the waveform sketch in Figure 14b would reach a negative peak voltage of —26 volts, as shown in the waveform sketch of Figure 13b. If S_1 were opened at $t_1 = \dfrac{\pi}{\omega_o}$ the capacitance

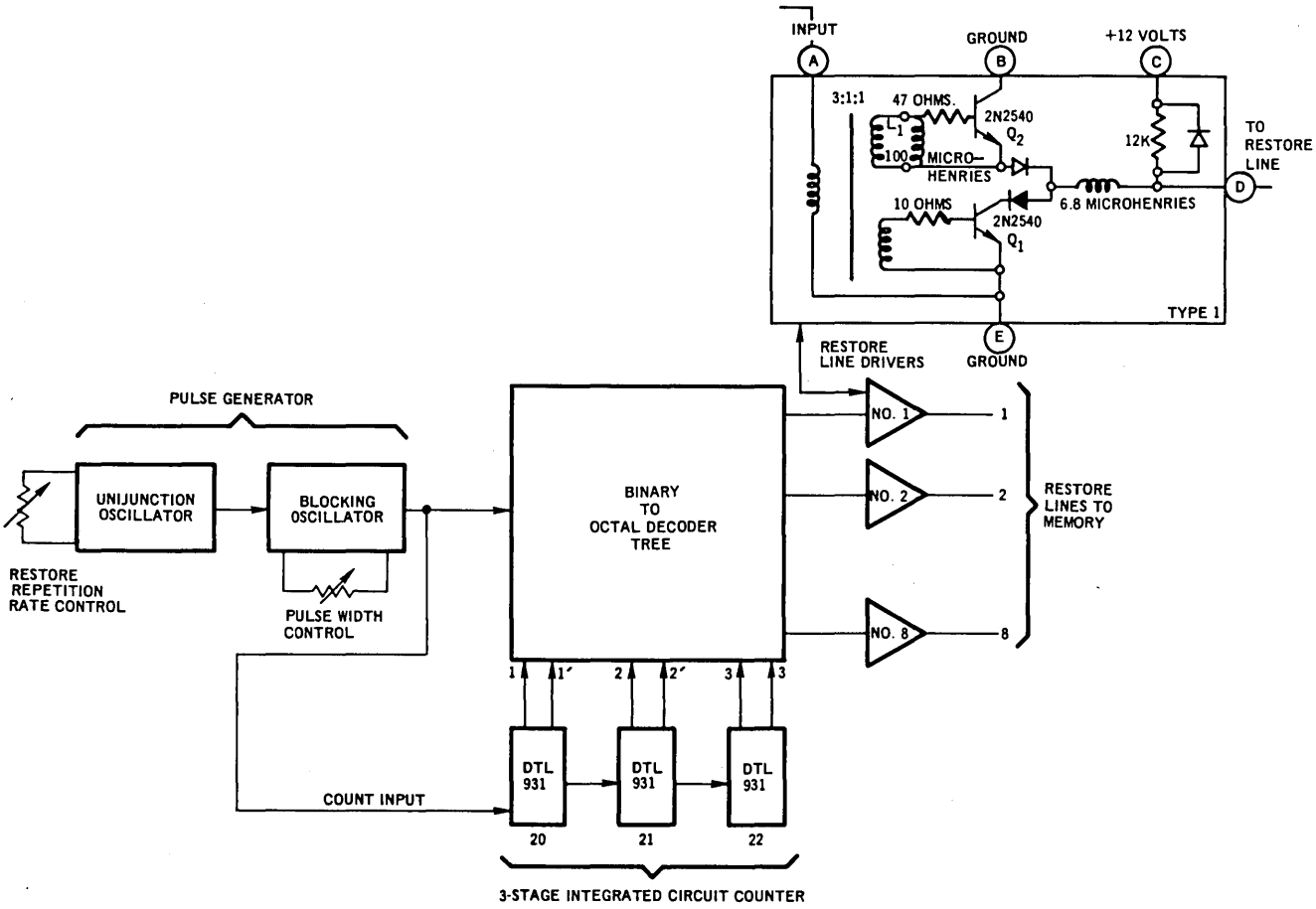


Figure 9—Restore generators using common pulse generator to supply restore line drivers

would remain charged to this value, with no power having been dissipated. If at time t_2, S_2 is closed while S_1 is opened, then the voltage on the capacitor is given by:

$$V_o = V_2 + [V_c(t_2) - V_2] \cos \omega_o t$$

where $V_c(t_2)$ is the voltage on the capacitor prior to closing switch S_2. In the lossless situation under consideration V_2 would equal V_1 in order that the trailing edge from t_2 to t_3 reach a peak value equal to the initial voltage on $V_c(O)$ prior to development of the pulse. With the ideal components that have been assumed the pulse has been developed without power dissipation. When component losses and voltage drops across semiconductor switches are considered, V_1 must be more negative than V_2 in order to accommodate the losses and develop the form of pulse shown in Figure 13b.

Practical application of this principle is illustrated in the case of the address driver by reference to the schematic insert shown in Figure 6. When the primary of the transformer is energized, Q_2 turns on as a saturated switch connecting the —7 volt supply through inductance L_1 to the capacitance of the address line. When the output voltage reaches the negative peak, diode CR_4 prevents reversal of current thus effectively disconnecting the output from Q_2. At the termination of the energizing pulse a high impedance is presented to the primary of the transformer and Q_1 is turned on by the voltage flyback action of the energy stored in the transformer primary[6] and in L_2. This causes Q_1 to turn on and charge the capacitor back to the initial voltage level. Resistor R_4 acts as a bleeder resistor to initially charge the capacitance to +12 volts. When circuits losses and semiconductor voltage drops are considered, the circuit parameters shown resulted in a 26 volt negative address pulse. With $L_1 = 2.7$ microhenries and $C = 1500$ picofarads, ω_o is found to be 15.8×10^6 radians per second, and the rise and fall time of the address waveforms is given by $t = \frac{\pi}{\omega_o}$ which is found to be approximately 0.2 microseconds. Typical X-Y address pulses developed by this circuit are shown in Figure 14.

This technique resulted in a considerable power savings in driving large capacitive loads. For example,

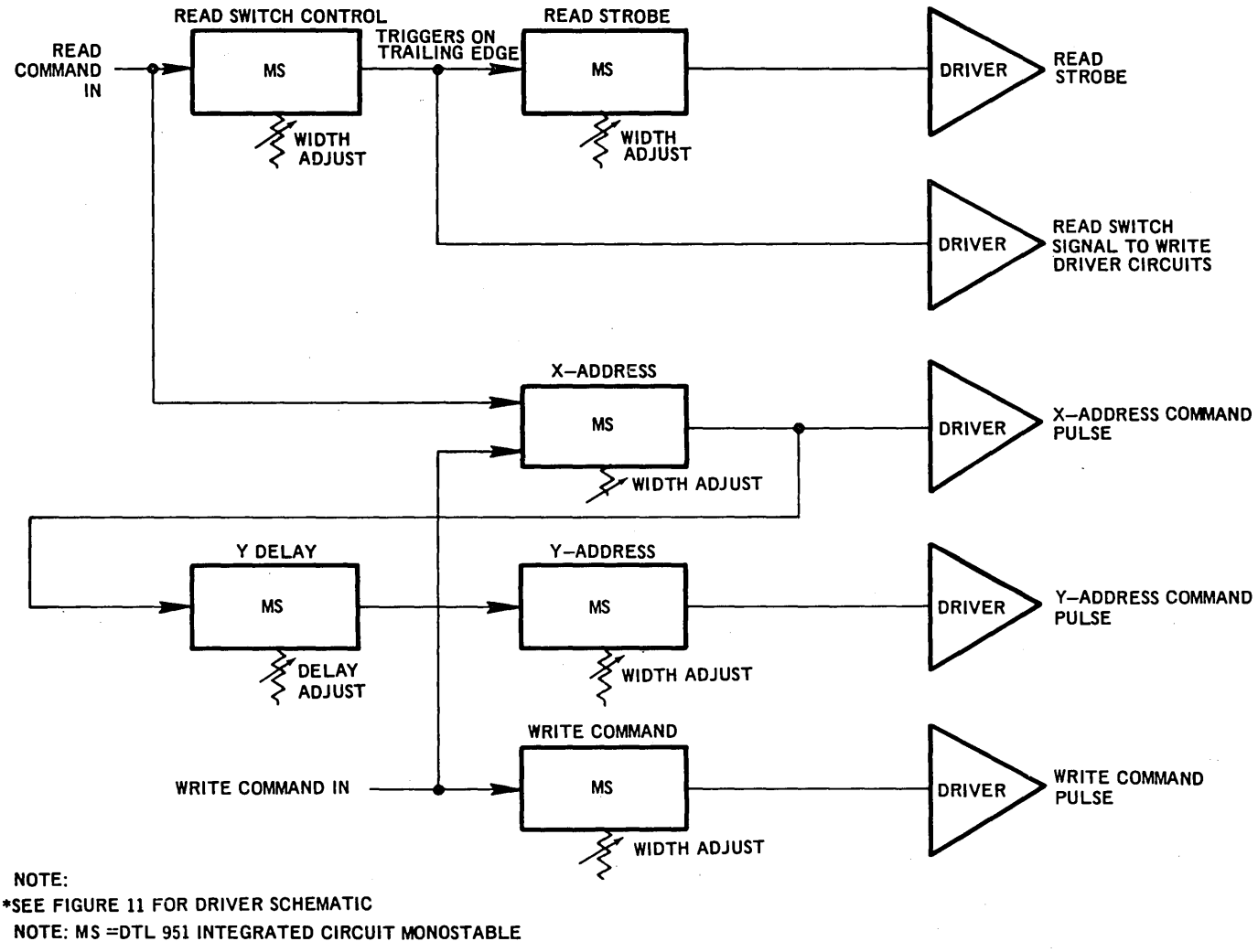


Figure 10—Pulse generator control circuitry block diagram

measured power for the two address driver circuits was about .7 watts, at a 1 mc addressing rate, whereas a resistive driver source would have consumed 1.6 watts in driving the line capacitance. The technique was successfully applied to the address and restore drivers; however, it was not applicable to the write drivers where power consumption due to transient charging of line capacitance was relatively high.

In the sense amplifiers discrete components were employed primarily because of suitable integrated form of low power sense amplifier was not available. Power conservation in the sense amplifier is achieved by use of high-resistance loads and a strobing technique previously described in which very little power is consumed in the circuit during quiescent (nonstrobed) operation.

Power requirements

The power objective for the memory proper was 2.5 watts for a 30% write-to-read ratio at 1 mc clock operation and for operation over a 0°C to 60°C temperature range. This power objective includes the memory stack and all of the peripheral circuitry except the address decoder. Power dissipation in the power supply and memory exerciser are not included in this figure. With 1.0 watts required by the address decoder for continuous addressing, the total power for the memory was to be less then 3.5 watts. Power actually measured was 2.4 watts, excluding the address decoders. With address decoders included, power was measured to be 3.4 watts. Of this the memory array itself consumes slightly less than 1 watt. With a con-

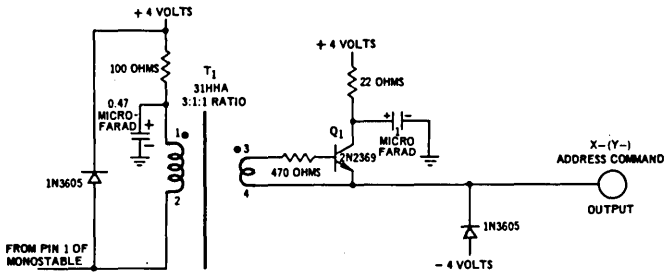


Figure 11—Address command pulse (X or Y) driver schematic

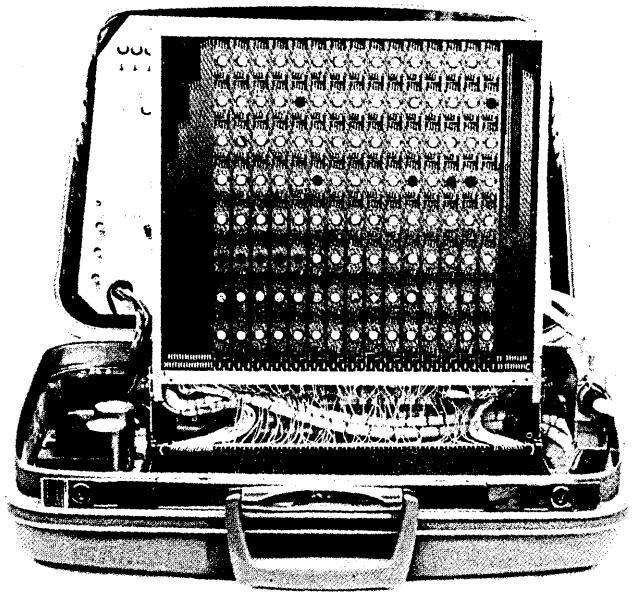


Figure 12—Suitcase configurations of memory and exerciser

certed effort to further reduce power dissipation, by design changes in the memory array and peripheral circuitry, it is estimated that the total power including the address decoding could be reduced below 2.5 watts.

Table III is a summary of the power dissipation in the various circuit sybsystems for 1 megacycle operation. The 30% write-to-read ratio results in the previously quoted power requirement of 3.4 watts.

Performance and evaluation

The entire system was made operational with the speed and power objectives achieved; namely (1) a two microsecond write-read cycle time, and (2) A total power consumption objective of 3.5 watts based on continuous 1 megacycle operation with a 30% write-to-read ratio.

Table III

Power Requirements for Memory Subsystems

	Write Continuous Operation (milliwatts)	Read Continuous Operation (milliwatts)	Standby (milliwatts)
(1) Monolithic memory cells	930	960	930
(2) Write driver circuits	2250	88	88
(3) Read sense amplifiers	120	450	120
(4) Control pulse circuits	240	240	210
(5) Restore generator	112	112	112
(6) Address driver decode circuits	1009	1009	480
Total power:	4661	2859	1940

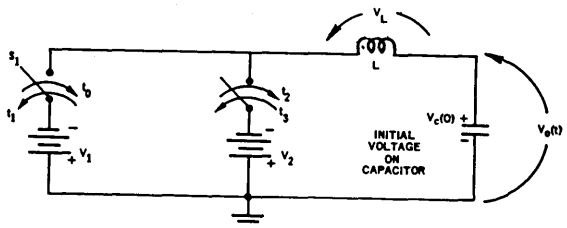


Figure 13a—Simplified schematic for line driver circuit

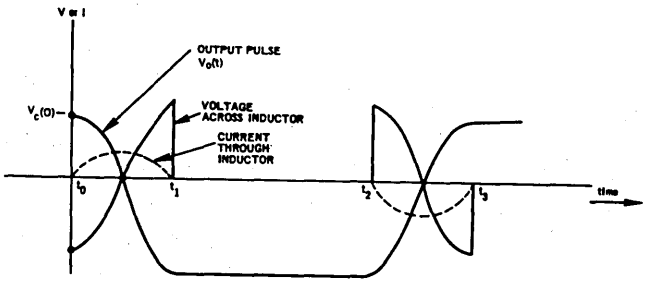


Figure 13b—Wave forms associated with operation of circuit in Figure 13a

In the initial design of the memory cell, a 5 KHz restore rate was estimated based on leakage measurements on sample units of MOS transistors. In order to accommodate the leakage variations in the 480 chips used in the memory, a 10 kc restore rate was found to be necessary. Although the entire system was not operated at 60°C, sample portions of the memory

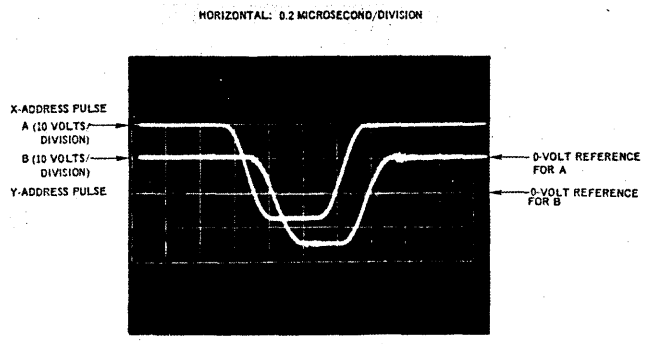


Figure 14—Address waveforms on X- and Y-address lines of memory

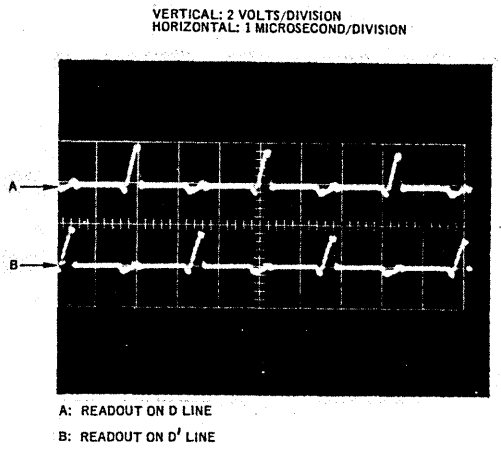


Figure 15—Readout with single flatpack connected to digit line

array were successfully tested for 60°C operation using the 10 kc restore rate.

Of primary importance to the peripheral driving circuitry are the line capacitances of the composite memory array. The measured line capacitances are shown in Table II. Various problem areas involving signal generation and distribution to the memory were anticipated and further investigated when the system was completed. Some of these are discussed in subsequent paragraphs.

To determine the effect of the increased capacitance and crosstalk on the digit line as increasing numbers of flatpacks are multiplexed together, readout waveforms were observed as flatpacks were added to one of the digit lines. Figure 15 shows the readout obtained with one flatpack (64 words); contrasted to this, Figure 16 shows the resultant readout with the full complement of 16 flatpacks, representing the 1024 words connected to the digit line.

Figure 17 shows the gross readout signal envelope over the scan of the 1024 words in the memory as

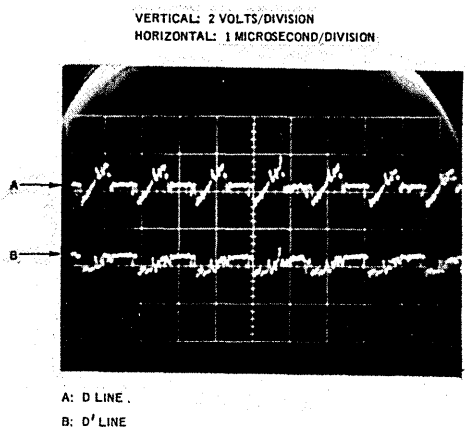


Figure 16—Read with full complement of 16 flatpacks connected

observed on one of the digit line pairs. A single logic "1" readout is shown on the $\overline{D}$ line for sake of comparison with the logic "O" level. The variation of readout amplitude over individual cells is vividly portrayed in this waveform envelope.

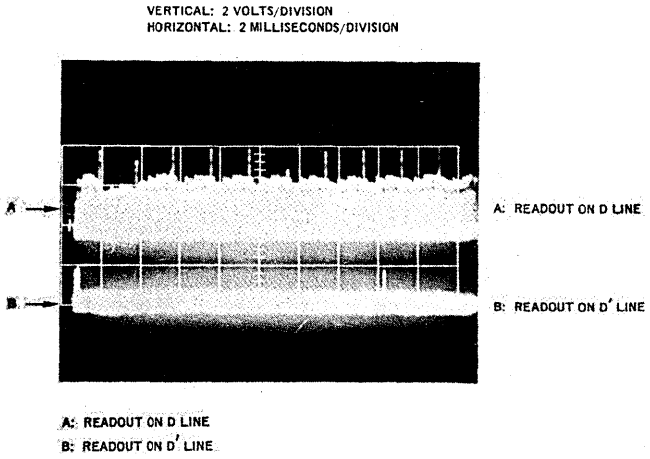


Figure 17—Comparison of readout envelope for the 1024 bits on the D and D′ lines

One concern in the readout operation was whether the crosstalk from the asynchronous restore signal on the flatpacks would be of sufficient magnitude to cause errors in the readout information. Crosstalk was found to result principally from parasitic capacitance coupling with the pulses utilized in the system. Address pulse crosstalk did not present a problem because these are synchronous signals and the readout strobe could be timed to minimize crosstalk from these signals. The

restore signal, however, being asynchronous, can present a "worst-case" situation when it occurs within the "window" of the read strobe. Fortunately, the use of a differential form of sense amplifier has the advantage of making only the difference in crosstalk between the D and D digit lines of importance. Because of a high degree of symmetry in the monolith and the printed board layout, this differential has been minimized. Figure 16, in the 4th waveform from the left, shows the effect of the restore crosstalk perturbation on the D and $\overline{D}$ readout lines with restore occurring at the peak of the readout signal.

To verify that recovery of the digit line after the 12 volt write pulse would be fast enough to allow correct readout response on a following readout cycle, the "write then read" mode was utilized while monitoring the waveforms appearing on the digit lines. Figure 18 shows typical write pulses followed by readout. In this photograph the write cycle is followed by three read cycles in order to make a comparison of succeeding readouts. It was verified that recovery was sufficient to allow a readout cycle to immediately follow a write cycle at a 1 MHz clock rate.

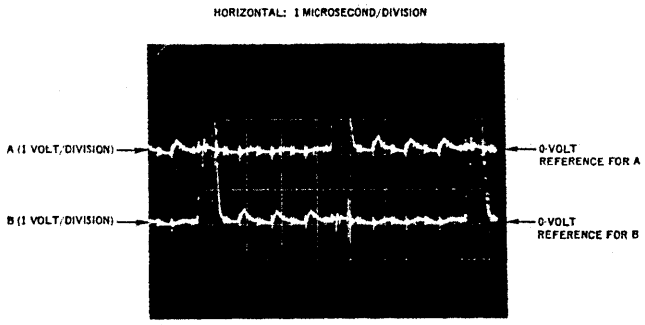


Figure 18—Digit line waveforms for write followed by read operations

A total of 480 memory chips was employed in the system, and thus represents a fairly sizable system using large-scale integration concepts. Although no quantitative reliability study was made as part of this program, after the debugging phases, the memory was operated over a long period of time without failures. However, the debugging phase was a difficult one primarily because of a number of intermittencies in several of the chips, which showed up very infrequently, and were not detected in the unit flatpack testing. The conclusion which can be drawn from this experience is that the unit test on LSI chips should be quite exhaustive if difficulties are to be avoided after the chips are installed in the system. Suitable testing techniques for LSI is presently one of the difficulties faced by the industry.

CONCLUSIONS

The low power MOS memory described in this paper represents a full scale implementation of a complete small-scale memory, not just a partially populated model. The approach has proven the feasibility of achieving low power through the use of pulsed operation of the memory array and peripheral circuits and by use of circuits capable of driving capacitive loads with reduced power dissipation.

The use of the restore technique to conserve power depends on maintaining control of leakage current, and is therefore susceptible to high temperatures and nuclear radiation effects. The present memory was designed for operating environments up to 60°C, and the effects of nuclear radiation was not considered in this program. Increased restore rates can be used for higher temperature and to accommodate leakage current increases by nuclear radiation but at the expense of greater power consumption.

In Table IV power requirements of the MOS memory are compared with those estimated for a number of other memory technologies assuming a 2048 word, 30-bits per word memory size. It is anticipated that power reduction improvements could be made in the MOS as well as other memory technologies by continued design effort with low power objectives.

Table IV

Comparison of Memory Technologies for Aerospace Applications*

	Speed (microseconds)	Power (watts)
MOS arrays	1 (NDRO), 2 (write)	3 to 5
Magnetic thin films	0.75 (DRO)	12 to 16
Plated wires	1.5 (NDRO)	10 to 15
Permalloy Sheet	2 (DRO)	7 to 10
Biax	1 (NDRO), 3 (write)	20 to 25
Magnetic cores	2 (NDRO) 2 (DRO)	15 to 20 25 to 35

*2048 words, 30 bits per word; equipment to meet MIL-E-5400 and to include complete electronics and power supply. 1969-1970 production assumed.
DRO = destructive readout; NDRO = non-destructive readout; NA = not available

The MOS memory in word sizes up to about 8000 words, does appear practical from the economic viewpoint in that the cost per bit is competitive with other memory technologies.

ACKNOWLEDGMENT

Grateful acknowledgment is made to the USAF Avionics Laboratory for their sponsorship of the Low Power Computer Memory Project.

The authors are indebted to R. Cole and R. Feuer for their contributions while at Bunker-Ramo, and to T. Kitaguchi for his role in low power circuit design. Walt Wittler's effort in fabrication and testing of the memory is greatly appreciated.

Finally, the authors wish to express their appreciation and acknowledge the efforts of R. Przybylski and D. Farina at the Philco-Ford Microelectronics Division for their role in the successful diffusion of the MOS arrays.

REFERENCES

1 R IGAROSHI T YAITA
An integrated MOS transistor associative memory system with 100 nanosecond cycle time
AFIPS Proceedings Spring Joint Computer Conference 1967

2 J R BURNS *et al.*
Integrated memory using complementary field effect transistors
ISSCC Digest of Technical Papers February 1966

3 I CATT E GARTH MURRAY
A high speed integrated circuit scratch pad memory
AFIPS Proceedings Fall Joint Computer Conference 1966

4 H PERKINS J SCHMIDT
An integrated semiconductor memory system
AFIPS Proceedings Fall Joint Computer Conference 1965

5 G PODRAZA S NISSIM D BREWER
A low power MOS monolithic memory array
Presented at the 1st annual IEEE Computer Conference September 1967

6 R LEE
Electronic transformers and circuits
Wiley & Sons New York 1955 pp 298-302

Development of executive routines, both hardware and software

by ALBERT B. TONIK
Univac, Division of Sperry Rand
Philadelphia, Pennsylvania

INTRODUCTION

Much has been written about Executive Routines (or Operating Systems) especially in the light of the present vogue for time-sharing and multi-processors. This present tutorial is an attempt to show the engineer what functions are contained in these executives. It starts with the most rudimentary form of executive and builds up to the most complex one. At all stages the executive system described is the most complex one for the biggest computer system. Smaller systems would leave out some of the functions.

It must be remembered that the user is seeing and is aware of less and less of the actual hardware. Figure 1 is a pictorial representation of this. The hardware does the actual processing of data plus auxiliary operations: I-O in parallel with computation, memory protection of one program from another, changing from relative addresses to absolute, timing the operations, etc. The Executive Routines keep track of all the programs that are scheduled to run, deciding which programs shall share the hardware system simultaneously, handling all of the I-O operations, handling error conditions and restarts, etc. The Assembly Routines allow the programmer to code in a language resembling machine code but easier to use and therefore less prone to stupid mistakes. The Compiling Routines allow the programmer to code a program in an English-like language so that many people can easily understand what the program is supposed to accomplish. The Generalized Application Routines allow a user to specify how his application differs from the general way of handling this type of problem.

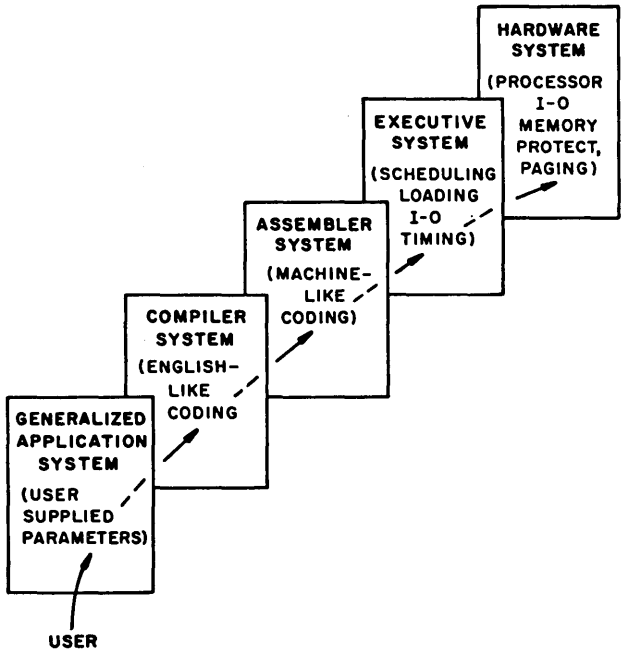


Figure 1 – The user's view of a data processing system depends upon his depth of knowledge

Simple executive

Bootstrap

Every computer designer has recognized the fact that programs have to be loaded into a computer even when the memory is blank. Therefore, every computer has been built with "Bootstrap" hardware that will allow the reading into memory of the first few instructions of a program. These instructions in turn will read the rest of the program into memory.

In the beginning the operators bootstrapped each program into memory. The operations people decided this was wasteful of computer and operator time. To reduce the unused time between programs and to reduce the load on the operators, programmers soon established the rule that each program should have a routine to duplicate the hardware bootstrap operation. At the completion of a program, the program would transfer control to the bootstrap routine in order to read automatically the next program into memory.

The programs were recorded on magnetic tape to accomplish the loading operation faster. But, it was

uneconomic to have only one program per reel of tape. So, many programs were recorded on a reel of tape. This led to the development of a few more executive functions.

Locator

One executive function when dealing with several programs on a magnetic tape is the locator routine. This is an enlargement of the bootstrap operation. At the end of a program, the locator routine is bootstrapped into the memory and the program that just finished transmits the name of the next program to be run to the locator routine. If the old program did not have a preferred next program and did not give a next program name to the locator routine, then the locator routine would ask the operator for the name of the next program. The locator routine has within it a table listing the names of all the programs on this program tape and where each program can be found. After the locator routine receives the name of the next program, it looks the program up in this table. Then it scans the program tape until it finds the beginning of that program. It bootstraps that program into memory.

The operator may have indicated that this next program is to be loaded from a different program tape or a different input mechanism in which case the bootstrap operation is performed for the other input device.

Another executive function is a routine that will perform various operations upon the program tape itself. This routine makes it possible to change any part of any program. It can resequence the programs on the program tape. It can add or delete any program to or from the program tape.

Linking loader

The load routines can become very complex. Initially a load routine was just a series of read commands to transfer a program from the input media into the memory. Then, later programs are loaded in pieces, where the pieces are subroutines that are scattered along the program tape. One program may be made up out of subroutines A, B, E, G, K, L, etc. while another consists of subroutines B, C, G, H, M, etc. See Figure 2. Therefore, when a subroutine is loaded into the memory, it will reside in different memory locations depending upon which other subroutines are loaded with it. Consequently, a subroutine cannot be written with absolute memory locations, but only relative locations. For example, a subroutine is written as though it would be loaded into a memory area beginning with memory location zero. Following the subroutine on the program tape is a table. This table indicates which memory locations within the subroutine have to be modified by adding

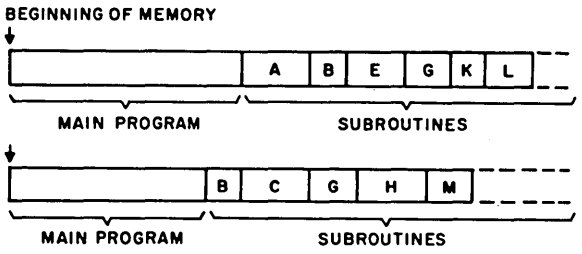


Figure 2 – Subroutines are relocatable because they reside in different memory areas with different programs

the number "n", where "n" is the actual location of the beginning of the subroutine, as it resides in memory within this program.

Memory locations are not the only quantities that are changed from relative to absolute by the load routine. Some subroutines will use a certain set of index registers, arithmetic registers, general registers, etc. The instructions in the subroutine will refer to these registers with a relative address. The load routine will indicate which registers this subroutine can use while running in this program. The table following the subroutine will indicate to the load routine how to change the relative register addresses to absolute addresses. Input-output channel numbers are handled in the same fashion, as is also the device number of an input-output mechanism connected to an input-output channel.

The load routine, which does all of the relativizing as explained above and one other function to be explained now, is called a "linking loader." The name implies that the subroutines have to be linked together to form a working program. Each subroutine can address specific locations within other subroutines and has specific locations within itself that other subroutines can address. For example, a multiprecision arithmetic subroutine would have locations within itself into which operands are to be deposited and different entrances for addition, subtraction, multiplication, and division. Some instructions within this subroutine might also address some locations within another subroutine, e.g., to move a multiprecision operand from one place to another.

Following each subroutine on the program tape is a second table listing all of the possible inter-subroutine connections for this subroutine. The load routine has to remember these tables for all the subroutines it is loading. After the subroutines are loaded, the "linking loader" changes all the memory references within each subroutine (which references refer to locations in other subroutines) to absolute memory locations by

using these tables. This operation cannot be performed as each subroutine is loaded because of "forward" references. A forward reference is a reference to a subroutine which is not in the memory yet and therefore, it is not known which absolute memory location to use when referring to a location within that subroutine.

One other function to further complicate the load routine is to change generalized programs into specialized ones Examples of this type of operation are the following:

1. A program is written to perform both a daily job and a weekly job. Inputs to the load routine specify whether the daily job or weekly job, or both, should be set-up this time.
2. Certain constants, such as today's date, have to be inserted into the program by the load routine.
3. Some programs are written to accept a multiplicity of inputs or a multiplicity of outputs. The load routine will change connectors within the program so that it will accept a specific number of inputs or outputs this time.
4. The assignment of channel numbers and device numbers to each input and output is done during load time.
5. The size of the block of data associated with input or output is established. Also, the number of buffer areas to be used for each input or output is set so that in addition to input-output operations overlapping computing, any large differences in the amount of computing per block can be smoothed out. This only happens when some of the computing is longer than I-O time and some shorter.
6. While attempting to debug programs, transfers of control can be inserted at appropriate points to enable a trace of the program to be established or to provide for selected memory dumps.

The inserting of the appropriate numbers into the program being loaded is accomplished by the same set of tables that are used for linking subroutines. The numbers (or constants) to be inserted come from operator type-in, or from a set of parameter cards, or from a previous program that calls this program into memory.

The load routine now occupies a big chunk of main memory. It is only used at the initiation of a program and at the initiation of an overlay if the program has such. This means that the memory occupied by the load routine cannot be used by the program. The size of this memory area locked away from the user raises a very legitimate question. Why not perform all the load functions during a prepass of the program to be loaded. That is, collect all the subroutines, changn them from relative to absolute, provide all the linkages, and specialize the program; then put the running program out on tape or drum to be loaded by a very simple load program. Unfortunately, this requires two passes of the program through the memory to ensure that all forward references are properly made and more I-O equipment is necessary in order to store the running version of the program. There is no clear cut decision as to which technique is best, and both have been implemented for different machines. For those utility programs that are loaded many times per day, the prepass conditioning is employed.

I-O handler

Some of the subroutines that are incorporated into a program are I-O handlers. The programmer no longer writes his own I-O commands within his program. Instead he transfers control to a subroutine with a request to retrieve or store a single word, a record, or a block of records. It is then the responsibility of the I-O handler subroutine to collect the requested data into or disperse it from a buffer. When the buffer is empty or full as the case may be, the handler has to issue an I-O command for that buffer. If the channel or device is already busy, the I-O command has to be queued to be issued at interrupt time. If, when the data request comes from the program, the buffer is empty or full so that the request cannot be honored, then the I-O handler does not exit back to the program until the data request can be satisfied.

When an I-O interrupt occurs, control is automatically transferred to an *interrupt subroutine*. This subroutine preserves a record of the state of the program that was interrupted. It determines which device caused the interruption and transfers control to the proper I-O handler. After the I-O handler has interpreted the interrupt and performed all the necessary functions, it transfers back to the interrupt subroutine. This subroutine then restores the computer to the state it was in just before interruption and transfers back to that place in the program where interruption occurred.

The I-O handler subroutine performs many functions as a result of an interrupt. If the interrupt indicates successful completion of the previous I-O command, then the subroutine makes the buffer available and issues the next I-O command if there are any in the queue. If the interrupt indicates an error condition, then the subroutine performs one of three functions.

1. The malfunction is such that if the program attempts to duplicate the last I-O command, there is a chance it might be successfully completed this time. These types of malfunctions are: parity errors while reading or writing on tapes or drums, punching errors while punching

data into blank cards, etc. If after several tries, the command still cannot be completed, it is considered to be a permanent error.

2. The malfunction is such that operator intervention is necessary before the I-O command has a chance to be performed successfully. This occurs when a tape servo loses power, when a card hopper is empty, or an output stacker full, when the printer runs out of paper, etc. The I-O handler subroutine has to determine the type of malfunction and transmit an appropriate operator message to the console handler subroutine. The operator can either fix the trouble or not. If he cannot, then he terminates the program. If he can straighten out the difficulty, then he automatically causes an interrupt by pushing an appropriate button on the peripheral device or else he types in a message on the console. As a result, control will be transferred back to the I-O handler subroutine, which will attempt to proceed with its queue of I-O commands.
3. The malfunction is such that the program cannot recover from the situation. In this case, the I-O handler subroutine transfers to an *abort subroutine* which prints out appropriate information for the maintenance men, throws the program off the computer, and waits via the locator routine for the operator to decide to return to a rerun point or go to another program.

The I-O handler subroutines have to be constructed to handle rerun points. First of all, one of the output subroutines has to establish rerun points. A rerun point is established by taking a static picture of the memory at some periodic interval, i.e., every 10 or 15 minutes.

When the instant to establish a rerun point has been reached, the output subroutine notifies all the I-O handler subroutines to come to an orderly halt and then notify the output subroutine. When everything is stopped, the output subroutine stores the entire internal memory on its output device. The memory contains a static picture of where the program is and where each I-O device is at this instant of time. The memory is outputted in a specific format, since it will exist in the midst of output data and has to be recognized as a rerun point and not data. After the memory is stored on the output media, the output subroutine notifies all the I-O handler subroutines that they can issue I-O commands again and then transfers control back to the worker program.

All input subroutines have to be constructed so as to recognize rerun points and skip over them. The input subroutine has to be alert, when reading the data serially, for information that is not part of the data, but is instead a rerun point. When the input routine recognizes a rerun point, it reads the entire memory dump and throws it away to get to the data on the other side.

In addition to the I-O handlers which establish rerun points, and bypass them, there also has to be a routine for initiating reruns. When the operator wishes to continue a worker program from some previously established rerun point, he has to load the *restart routine* via the locator routine. The restart routine then asks the operator on which I-O device it can find the memory dump which established the rerun point and the identification of the particular rerun point. (As each rerun point is established and outputted into an output file, the necessary identification is printed on the operator's console.) The restart routine then scans the data on the I-O device until it locates the correct memory dump. It examines this static picture of the previous memory contents.

The restart routine has to position all the I-O devices to exactly the positions they were in when the rerun point was established. To accomplish this it has to look at a fixed position in the memory dump to find out which I-O handler subroutines were used by the program and where the subroutines are located in the memory dump. It then asks the operator if any channel or device numbers have to be changed. After the numbers have been verified by the operator, the restart routine forces the I-O handlers to read the beginnings of the files on all the devices and check the identifications of all files. This is a further check on the correctness of the operator. Then the restart routine forces the I-O handlers to skip past each file's data until each file is aligned with its position as indicated in the memory dump. (Each handler has a count of where it is within the file or else the data in the buffers has to be compared with the file data.) When this point is reached, and the rerun point memory dump is loaded into the memory, the restart routine finally turns control over to the worker program at the point where the output subroutine established the rerun point.

The I-O handler subroutines *cannot* address the I-O device directly with channel and device numbers whenever the worker program addresses a file. The reason lies in the restart procedure just outlined. When a program is initially loaded, specific device and channel numbers are assigned to each file that the program will address. However, when a program is restarted at a rerun point, these files may be assigned to a different device or a different channel (a tape handler may have broken down or a channel gone out of commission). Since programs can modify themselves, it is no longer known which memory locations

have to be modified in order to change a device or channel number.

Therefore, a table of file numbers and device numbers is constructed in memory. This table has an entry for each file that the program will address. Each individual entry will contain the device and channel number of where the file is located at this time. This table is accessed by each I-O handler subroutine. Whenever an I-O handler subroutine is about to issue an I-O command for a particular file, it looks up in this table the device and channel number for this file and inserts them in the I-O command which is then transmitted to the I-O device. The entries in this table have to be updated by the restart procedure according to instructions from the operator. Some of the entries in the table are rotated when the program switches from one reel to the next of 2 multi reel file.

Multiprogramming

Now, the situation involving the slower speed peripherals has to be examined in more detail. At first there were tape oriented processors and the slower speed peripherals were handled by off-line devices. There was an off-line card-to-tape converter, a tape-to-printer converter, etc. Then it was decided that it would be cheaper and more flexible if there was a small, satellite computer to which all the slower speed peripherals were connected. In the satellite, one program would perform a card-to-tape conversion, then another would perform a tape-to-printer conversion, etc. Then it was decided that it would be still cheaper if the slower peripherals were connected directly to the main processor. But now, it was noticeable that when a program included a slower speed peripheral, most of the time the processor was sitting idle waiting for that peripheral to finish an I-O command.

Thus was born the idea of multiprogramming (several programs should be run in an overlapped fashion to maximize the utilization of main processor time). This means that a computer-limited program will only require a little additional time to run on the computer system when running with a peripheral program. However, both programs will run in this time compared to twice the time when run separately. In addition, several peripheral programs could be run in the time ordinarily required for just one of them. I have never seen any proof that the extra computer time required to run the peripheral programs made this system more economical than the extra hardware of a satellite system to perform the same job. Nevertheless, the computer field has gone this way. (The time-sharing systems finally seem to be justifying this approach.)

The concept of multiprogramming has caused a giant step in the number of functions to be included in the executive system complex. The end of these new functions has not yet been reached since multiprocessors, real-time and time-shared systems are just variations on the theme of multiprogramming.

When there was only one program in the memory at a time, the program tended to use all available memory. The available memory was not the entire memory because a certain part of the memory was occupied by bootstrap, load, restart, trace, etc. routines. If the program did not fit into memory, then the programmer had to decide how to break the program into overlays (segments) or into different runs. In a multiprogramming environment, *a program must use memory efficiently.* Therefore, the programmer has to decide which part of the program has to be in memory at all times so that the program will run fast. The other parts of the program which are only used occasionally have to be divided into overlays. On the rare occasions when one or more of these parts of the program are needed, they would be located and loaded into an overlay portion of the memory space occupied by this program. Handling programs in this fashion means that a maximum number of programs can be stored in memory at any time.

Another device for the conservation of memory space is the use of *common subroutines and re-entrant subroutines.* A common subroutine is one which can be entered from one of several different programs, perform a function, and then return control to that program. A re-entrant subroutine is also one which can be entered from one of several programs. In addition, while in the subroutine, the program can be interrupted and control transferred to another program which in turn can enter the re-entrant subroutine. This means that several different programs may be in the subroutine simultaneously. The rule to make this possible is that a re-entrant subroutine is never allowed to modify itself. Therefore, any variable quantity (instruction or data) referred to by the re-entrant subroutine has to reside back in the program area of the program that entered the subroutine. These quantities are referred to by index register and indirect addressing. A variation of the re-entrant subroutine is a *recursive subroutine* which allows the same program to call on it in a nested sequence. To use a recursive subroutine there is a pushdown list of the contents of the index registers and indirectly addressed locations, which contents refer to the variable quantities stored in the program area. Before the program transfers control to the recursive subroutine, it pushes down into the list these contents and then loads the index registers and indirectly addressed locations with the references to be used

this time. When the recursive subroutine returns control to the program, the program pulls up out of the list, the previous contents and restores them into the registers and locations. An example of a common subroutine is an I-O handler subroutine, while an example of a re-entrant subroutine might be an output editing subroutine.

Every time a program is run it will probably be run with a different combination of programs. This means that each time it is run it will probably reside in a different area of memory. Therefore, each program must be written in a relocatable manner. As described previously, subroutines were relocatable, but not necessarily worker programs. Now however, even the *worker programs have to be relocatable.*

Locator or scheduler

The locator routine in a multiprogramming environment has extra functions. When a program is finished, the locator routine is loaded. The *locator routine has a list of programs waiting to be run.* This list is arranged by priorities and within each priority by when the request was made for this program. If the program that just finished, calls on another program and leaves information for that program, the locator has to remember this data for initializing the called program when it finally is loaded into the memory.

The locator routine extracts a program name from the waiting queue. From the highest priority list it takes the oldest. Stored with each program is a list of facilities that the program needs. The list includes the amount of memory, the type of I-O devices, and the quantity of each device. The locator routine checks for these facilities, i.e., are they in the computer system and are they available (no other program is using them at the moment). If the facilities are available, then the locator routine asks the load routine to load that program. If they are not available, the locator routine goes to the next program waiting in that priority list and tries again. If there are no more programs under that priority, it goes to the next lower priority and tries, etc.

The locator routine now has to maintain allocator tables. The allocator tables indicate which facilities have been allocated to which program. There is an allocator table for the memory which shows which area of memory is assigned to which program and which areas of memory are free (not assigned to any program). If there are auxiliary memories such as drums or discs, then there is an allocation table for them. Finally, there is an allocation table for all the I-O devices. This last table may be a different one than that one which changes a file number within each program into an absolute device number. Both tables can contain the same information but for fast look-up of table contents, each of these two applications requires a different sequence within the table.

The operator has the option of changing the queue of programs at any time. He can type a message which asks for the locator routine. When the locator routine gets into memory, it provides the operator with a number of options.

1. The operator can ask for everything to stop immediately because the end of the shift has been reached and the system will be turned off.
2. The operator can ask for a specific rerun of a specific program. The locator has to determine which memory area the program occupied originally. It does this by looking up the information in the memory-dump-rerun point. Then it sets this memory area aside so that no other program can be loaded into it. When that memory area is free, the locator transfers control to the restart routine.

   The rerun point has to be loaded into the same memory area that it originally occupied because it is not known how a program will modify itself during running and therefore it would be difficult to relocate it.
3. The operator can add a new program to the queue. He can type in the name of the program and all the parameters to set it up. Or he can indicate on which input device the list of parameters can be found. Or he can indicate on which input device the list of parameters and the program can be found. In any case, all the inputs are read and stored on a tape or drum until needed. This frees the input device for use by another program.
4. The operator can change the priority of a program that is in the queue.
5. The operator can delete a program from the queue.

During the execution of these locator functions the central processor is still being time-shared with all the programs in memory.

Memory protection

With more than one program in the memory at any time, the question of protecting programs from each other naturally comes to mind. Each program wants assurance that some other program will not suddenly run wild and destroy something within this program's memory area. Also, a program sometimes wants assurance that no other program can peek over and examine some confidential data to which it alone should have access. Finally, a constant surveillance must be maintained to insure that I-O commands only

transmit data into or out of permissible memory areas and on or from permissible I-O devices.

Memory protection has been handled by a combination of hardware and software. The locator routine maintains allocation tables. Whenever control is transferred from one program to another in memory, the memory area for that program is extracted from the allocation table and placed into a special register by means of a special instruction. This special instruction insures that a worker program cannot change the contents of this special register. Whenever a memory reference is made by that program, the hardware automatically checks that the memory location lies within the prescribed memory area. When an I-O command is handed over to an I-O handler subroutine, the handler checks the indicated buffer to determine if it lies within the allowed memory area. Just before executing an I-O command, the handler will check the I-O device allocation table to make sure that the program can address the device and the area on the device. Then the handler converts the relative device and channel numbers to their absolute equivalents.

There is a set of special instructions which cannot be executed by a worker program but only the executive routine. These instructions deal with memory protect, I-O commands, interrupt, and transfer between programs.

This method of memory protection makes it a little awkward to handle common subroutines and reentrant subroutines. These subroutines are not included in any one program's memory area, since they are shared by several programs. When a program is preparing to use one of these subroutines it creates a list of parameters within its own memory area, and then transfers control to the subroutine via a special instruction. This special instruction has to perform a number of functions simultaneously. First of all, only one worker program is allowed to use a common subroutine at any one time. To ensure that this condition will be true, a sentinel is set within the common subroutine's memory area whenever a worker program is using it. The special instruction checks this sentinel: if not set, the instruction sets it and allows the worker program to transfer control to the common subroutine; if set, the instruction causes the worker program to give up control of the central processor in the same way as an I-O interrupt will do. The entire common subroutine including this sentinel is outside of the memory-protected area of the worker program and therefore, the special instruction has to disable the memory protect hardware. Finally, the special instruction transfers control to the common subroutine and checks to ensure that this transfer address is outside of the memory-protected area. The subroutine itself has *no* memory restrictions, so that it can get to the list of parameters and store the results in the same area. The subroutine returns control to the worker program via another special instruction which reactivates the memory protection feature and resets the sentinel.

I-O handler

One type of common subroutine is the I-O handler subroutine which handles more than one device on a control unit. Examples of these devices are: mass storage, tape units, and multiplexed terminals. The handler subroutines for these devices consist of two parts. One part is duplicated in each worker program using the device and handles the individual records within a buffer.

The *other part is a common subroutine* and handles the transmission of data between the buffers and the I-O devices. This common subroutine accepts I-O commands from the worker programs. It forms queues of these commands (when they arrive more rapidly than they can be initiated) by program priority. It performs the memory protection operation on them. It issues the I-O commands when it can, and inserts absolute device addresses. At interrupt time, it analyzes the results for successful or unsuccessful completion. It attempts to recover from error situations. After successful completion, it sets sentinels in that part of the I-O handler subroutine within the worker programs so that they can use the proper buffers. If the command cannot be completed without error, then it communicates with the *jettison routine* (which removes the corresponding worker program from the memory and allocation tables and queues of I-O commands).

In some systems, the I-O handler subroutine compiles amount of time used on the device or channel so as to be able to compute charges for each worker program.

Dispatcher

The nerve center of an executive system is the dispatcher routine (an extension of the interrupt routine for single program control). When an interrupt occurs, the dispatcher routine must capture a picture of the condition of the interrupted worker program. It must preserve the state of all flip-flops and registers so that the worker program can continue from that point at some later time. It stores all of this information in a part of the memory area assigned to that program. Then the dispatcher routine determines the cause of the interruption and transfers to the appropriate I-O handler subroutine. After the handler subroutine

processes the interrupt (notifies the worker program of completion, starts another I-O command, etc.), it transfers back to the dispatcher routine. If some other interrupt is waiting for processing, the dispatcher routine will transfer to the appropriate I-O handler subroutine. When there are no more interrupts to be processed, the dispatcher routine will use a predetermined algorithm to determine to which worker program control will be transferred. The algorithm might be

1. Always go back to the interrupted program
2. Always go to the next program
3. Spend an amount of time in each program which time is in agreement with the priority of the program
4. Or any other reasonable one.

To return control to a worker program, all the flip-flops and registers have to be restored to the state they were in when the program was interrupted.

Part of the above algorithm is to skip over programs which have relinquished control voluntarily. A program will give up control voluntarily for one of the following reasons.

1. It requires the use of a buffer for which an I-O command was issued, but not completed.
2. It attempts to issue an I-O command, but the queue in the I-O handler is full and the I-O handler will not accept it.
3. It is establishing a rerun point.
4. It is waiting for an overlay.

These conditions are recognized in the worker program part of the I-O handler subroutine. Since this is a library subroutine called on by worker programs, it is never left up to the programmer to voluntarily release control. It is built into the "canned" subroutine used by the worker program. When an I-O handler subroutine, which is included in a worker program, realizes that it cannot proceed, it transfers control to the dispatcher routine.

The dispatcher routine will temporarily remove that worker program from the list of programs amongst which it apportions central processor time. The dispatcher routine also makes a note of which interrupt will reactivate the worker program. Then the dispatcher routine transfers control to some other worker program. When future interrupts are detected, the dispatcher routine (in addition to its usual functions) has to decide if one of the suspended worker programs should be returned to the active list.

The dispatcher routine also provides for totaling the amount of central processor time used by each worker program. In addition, it provides for time interrupts so as to insure that one worker program will not use more than its allotted share of central processor time.

Rerun

Before establishing a rerun point by taking a picture of the memory all I-O commands have to be completed. When only a single program was running on a computer, only the I-O commands that had been initiated had to be completed. In a multiprocessing environment all the I-O commands in the queues within the I-O handler subroutines have to be initiated and completed. The reason the queues have to be emptied is so that when a program is restarted at a rerun point, and the I-O handler subroutines are also reloaded, there will be no I-O commands within the I-O handler subroutine which commands are for other programs that existed when the program was run and not now when it is being restarted.

Associated with each program in the memory is a list of the common subroutines that that program is using. This list has to indicate which common subroutines are being used and where they are located. This information is necessary for reruns. First of all, the rerun memory dump has to ensure that the program area and all the areas occupied by the common subroutines used by that program are included. To restart the program at a rerun point, the program and its common subroutines have to be reloaded into their original memory areas. The freeing of these memory areas is checked by the locator routine at the finish of each program. The locator routine also has to keep track of how many programs are still using a common subroutine. When none of the programs left in the memory want a common subroutine, the area occupied by that common subroutine becomes free. The locator routine also has to guarantee that certain common subroutines, such as I-O handlers, *cannot* be duplicated in the memory. Therefore, a program cannot be restarted as long as one of its I-O handler subroutines is in memory, even though the memory occupied by the subroutine does not interfere with the loading of the rerun point. The program to be restarted has to wait until the programs which are now using those I-O handler subroutines are finished and disappear from the memory.

Paging

All of the above constitutes the executive system for a multiprogramming system. Now people are trying to eke a little more economy out of the system. It is noticed that sometimes, all of the central processor time is not used. This occurs when all of the programs in the memory are input-output limited. In this case there is some percentage of central processor time not utilized. Also, when one of the programs is computer limited and the others input-output limited,

then the unused central processor time will occur from the time that the computer limited program finishes until the time that the next computer limited program is loaded and ready to go.

The way to utilize more central processor time is to have more programs in the memory. But the question is: how to put more programs into the memory without enlarging the memory. One observation is that there are many small memory areas scattered throughout the memory that are not used by any program.

These empty memory areas occur in the following way. When a program finishes, the memory area occupied by that program is now vacant. The next program to be loaded into that area will undoubtedly be smaller, since it has to fit into that area. This leaves a small vacant area. Also, if the program that finished was the last program in the memory at this time to use a common subroutine, then the area occupied by the common subroutine is also vacant. This area may or may not be filled by the next program requiring a common subroutine which will fit.

The obvious solution is the ability to break a program into pieces to fill the available memory areas. The linking load routine could handle this situation with the additional function of being able to insert unconditional transfers so that the program could jump from one area to another. However, the memory protect scheme becomes very complicated. To simplify the hardware required to handle pieces of program scattered around the memory, the memory is divided into fixed sized "pages." Every time the memory is addressed by a program, the memory address is separated into two parts: an address of a page within this program, and an address of a location within this page. The page address of a page within a program is transformed by means of a table which contains the absolute memory location of where that page actually resides in memory this time.

This paging scheme produces a change in many functions of the executive system. The load routine, when it collects all the subroutines into one program, changes the relative locations to those that are relative within the program and not to absolute memory locations. A program is written as though all of it were in memory at once. A programmer does not worry about overlays, but he does try to get all parts of the program that are used together in one compact area. This means that when a cohesive group of parts is in the memory there will not be wasted memory containing parts of the program not needed in memory at this time. (*Memory protection* now means that a program cannot address anything outside of its own area (this area is the total size of each program). Memory protection will now cover the common subroutine. When a worker program transfers control to a common subroutine, the memory protection afforded by the page table will only allow the common subroutine to address locations within itself or in the worker program area.

The table that transforms relative page addresses to absolute memory addresses performs other functions. It determines whether the page is in main memory or not at the moment. It can indicate whether a page has been written into while it was in main memory. If so, the updated page has to be preserved on back-up memory if the page has to be removed from memory to make room for some other page and the removed page is to be used later on in the program. It could also indicate how recently the page was used. This information is used to determine if this page could be disposed of when a new page has to be brought into memory.

Many systems are advocating that the hardware automatically handle the determination of which pages are in the memory. They have the rule that no page is brought into memory until it is actually addressed. They invent complicated algorithms to determine which page should be thrown out to make room for this new page. The reasoning that justifies this approach is that only a small part of any program is actually used during any small period of time. On examining running programs and the memory occupied by them, people have estimated that only ten to forty percent of the program is used during any short interval of time. The feeling is that the programmer is not smart enough to load into memory that part of the program that is needed at this time. Also, that he will cheat and ask for too much of the program to be in memory. On the other hand, asking for a page at a time is inefficient because of access time. It would be much faster to transfer a group of pages at once. Also, the algorithm which decides which page is no longer needed and gets rid of it, may not realize that the program is not really finished with that page and will address it shortly.

The programmer should indicate which parts of his program should be in memory simultaneously. This has to be accomplished within the programming language that he uses. He does this realizing that if he asks for too much memory, he will slow down his program because

1. there will be times when his program cannot be loaded,
2. there is a greater possibility that his program will be kicked out of memory before it is finished by some other program. Therefore, there will be wasted time to reload his program before it can continue.

The decision as to which programs are to share memory at any one time is very complex. The queue of programs which are still to be run is maintained by the scheduler routine. This queue is in priority sequence and the priority of a waiting program can be raised as time advances. New programs are initiated when memory space becomes available or when the priority of the program is above a certain threshold. When a new program or a new section of a present program is to be loaded and there is no room in memory, then one of the programs in memory is thrown out. The decision of which program to delete is based on program priority, how much computer time the program has used, and how much memory space it occupies. Before a program is tossed out of memory, all the I-O commands issued by that program, which commands are still in the queues of the I-O handler subroutines, are allowed to be completed.

Common subroutines

Common subroutines fit into the paging scheme very conveniently. A common subroutine is coded relative to memory location zero. It is loaded anywhere in the memory in that form. Any memory references made by the common subroutine are augmented by at least one of two index registers. These registers were loaded by the worker program that transferred control to the common subroutine. One index register contains the location within the worker program of where the program transferred control to the common subroutine. This index register helps the common subroutine address the list of parameters. The other index register contains the location of where the worker program thinks the common subroutine is located within the memory area of the worker program. This index register allows the common subroutine to address any location within itself. All of these memory references are translated by the program page table from locations within the program area to absolute memory locations. Therefore, each program that uses a common subroutine has entries in its page table for that common subroutine.

There are additional hardware controls governing the use of common subroutines. Only one program can be in a common subroutine at any one time. That program has to exit from the common subroutine back to the program before another program is allowed access to the common subroutine. There has to be a sentinel within the common subroutine indicating whether or not a worker program has transferred control to the common subroutine. A worker program has to issue a special instruction to transfer control to a common subroutine. This special instruction checks the sentinel and if it is reset, sets it and allows the transfer of control and possibly may fill the index registers. At the same time a flip-flop in the status of that program is set, indicating that that program is in a common subroutine. The common subroutine has a special exit instruction which resets both the sentinel and status flip-flop. The memory protection afforded by a program's page table prohibits a worker program from writing into the area occupied by the common subroutine until control is transferred to the common subroutine. This write protection is necessary because some other program may be within the common subroutine.

When establishing a rerun point or when a program is temporarily tossed out of memory, the program *cannot* be within a common subroutine. The status flip-flop helps to insure this condition. The reason is that other programs may want to use that common subroutine in the meantime. Also, when the program is brought back into memory, it cannot be within a common subroutine because the common subroutine is not brought into memory if it is already there (maybe in a different place), since it is being used by some other programs.

The executive routine still is required to maintain a count, for each common subroutine, on the number of programs now in the memory that require that common subroutine. When such a count goes to zero, the executive routine knows that common subroutine is no longer needed. It assigns the pages of memory occupied by the common subroutine to the free memory part of the memory allocation table.

One type of common subroutine, the *I-O handler subroutine,* requires additional hardware. There is only one copy of the common part of an I-O handler subroutine in memory. All I-O commands for that device or group of devices handled by the subroutine have to be queued in the I-O handler to be issued serially. Each I-O command has a buffer location expressed as a relative location within a program's memory area. When the I-O command is sent to the channel for execution, the address of that program's page table has to be sent in addition, so that the channel can determine absolute memory locations of words in the buffer. To obtain the address of a program's page table requires a special instruction which can only be executed within a common subroutine.

This same special instruction is used at interrupt time. Right after interrupt time, control is transferred to the I-O handler subroutine. The handler subroutine determines whether the I-O command was completed without error. If so, it has to notify that part of the I-O handler subroutine which is contained in the worker program. To do this, the common part of

the I-O handler subroutine has to have access to that program's page table. Also, the dispatcher routine, when transferring control to a program, uses this special instruction to notify the memory addressing mechanism of which page table to use.

Rerun

To establish a rerun point, the first thing that has to be done is to throw the program out of memory. To accomplish this, certain rules have to be obeyed. The program cannot be in a common subroutine (this is not true of a re-entrant subroutine.) If it is, the program has to keep running until it is no longer in a common subroutine. Then, all I-O commands which are queued up in I-O handler subroutines have to be completed. Then the image of the program which is recorded on back-up memory is updated. Then a copy is made of this program image and given a restart identification (a memory dump is no longer needed).

The operator has to reinitiate a program from a rerun point. When the scheduler routine decides to allow this program to be reloaded, it locates the proper program image. The program status indicates which pages are to be loaded into memory. The program image does not contain common subroutines, but only a list of which are needed with that part of the program loaded in memory. The scheduler routine checks its list of common subroutines that are prescently in memory. If any are needed that are not presently in the memory, then the locator routine loads these common subroutines also. Finally, a page table is constructed for this program.

Multiprocessors

The final changes within the executive system are due to multiprocessors. There are two reasons for having more than one processor in a configuration.

1. Supposedly with a paging scheme and with larger, "cheaper" memories, many more programs are sharing the central processor and the central processor is heavily computer limited.
2. The "graceful degradation" idea is familiar to all of you and needs no further explanation.

Dispatcher

One of the functions of the dispatcher is to see that each processor gets to perform its share of the work load. The dispatcher routine as well as any worker program has to be able to run in any of the processors. However only a re-entrant subroutine can be executed by more than one processor at a time. When a processor switches from one program to another, it does this under control of the dispatcher routine. To perform this switching job, the dispatcher routine should run in this processor (rather than interrupt some other processor while this one stands idle). Therefore, the dispatcher routine (when it is needed) should be run by whatever processor is handy at the moment. However, there has to be memory protection hardware which will prevent any other processor from entering the dispatcher routine while it is being run since certain parts of the dispatcher are *not* re-entrant.

There are many reasons for transferring control to the dispatcher routine.

1. An I-O interrupt has occured.
2. A program voluntarily gives up control.
3. The allotted time period has expired for the running of a program.
4. The operator wishes to start a processor which has been idle.
5. An error has been detected.
6. A processor has gone down.

The decision as to which processor will be interrupted to transfer control to the dispatcher routine is based on

1. A processor number designated by the interrupt or
2. which processor has the lowest priority program or
3. which processor is running the program that was the cause of the interrupt.

Real-time interrupts cannot afford to waste time searching for a processor to interrupt, and therefore, they will interrupt a specific processor. If a processor cannot be interrupted because it is down, then some other processor must be interrupted.

The dispatcher requires special instructions to communicate with other processors. The dispatcher routine needs to test the state of the other processors.

1. It is available and running.
2. It is available, but idle.
3. There is a detected error.
4. It is unavailable) power is turned off, or it is offline for maintenance testing).

There has to be an instruction which will start a designated processor at a specific memory location. There has to be an instruction to test for any interrupts which have not been satisfied. While one processor is in the dispatcher routine, other interrupts can come along but do not cause any other processors to enter the dispatcher routine because only one can be within that routine at any one time. There has to be an instruction which will stop a designated processor because the time interrupt was recognized in some other processor. If a machine error is detected while a processor is in the dispatcher routine, then all processors are halted and the operator has to restart all the programs.

Because more than one processor is operating simultaneously, different parts of a single program may be performed simultaneously. This is already true in a single processor system (in a restricted sense), since I-O commands are performed simultaneously with computing. To perform computing simultaneously in more than one branch of the program, the programmer has to indicate that certain branches are independent of others. Special instructions are needed in the program at branch points so that an independent branch could be initiated, if there was an available processor. Eventually in the program, a point is reached where independent branches all have to be completed before the program can proceed. At this point another special instruction is needed to keep count of the completed branches.

If there are independent branches of the program, then there has to be some intelligence in the control which knows the maximum number of parallel branches that can exist at any time and how many are being used at any moment. For each possible parallel branch there has to be an area set aside in the program memory for storing the status of the program in case it is interrupted while in that branch. Within the dispatcher routine the list of programs, that are awaiting service by some processor, has to have entries for each branch that is in the midst of being executed. This entry also has to know where to find the status of the program so that it can continue from the point of interruption. If a program is ever restarted from a rerun point, then the *restart routine* must know how many branches to reactivate and where to find the status of each one.

Common subroutines

The control of a common subroutine is handled in the same fashion as indicated before. Only one program is allowed in a common subroutine at any time. Because more than one processor is running, "race" conditions have to be guarded against; that is, two processors trying to transfer control to one common subroutine at almost the same instant of time. Therefore, one instruction has to test the common subroutine sentinel, set it if it is reset, transfer control to the common subroutine, and allow the common subroutine to write into its own area.

BIBLIOGRAPHY

F L ALT
A Bell Telephone Laboratories computing machine
M T A C No 21 page 1 1948
F E SNYDER and H M LIVINGSTON
Coding of a laplace boundary value problem for the UNIVAC
M T A C No 25 page 341 1949
R K RIDGEWAY
Compiling routines
Proceedings of ACM page 1 Sept 1952
H B DEMUTH J B JACKSON E KLEIN N METROPOLIS W ORVEDAHL and J H RICHARDSON
MANIAC
Proceedings of ACM page 13 Sept 1952
M V WILKES
Pure and applied programming
Proceedings of ACM page 121 Sept 1952
N ROCHESTER
Symbolic programming
PGEC Vol 2 no 1 page 10 1953
R A BROOKER and D J WHEELER
Floating operations on the EDSAC
M T A C No 41 page 37 1953
W G BOURICIUS
Operating experience with the Los Alamos 701
EJCC page 45 1953
J N P HUME
Input and organization of subroutines for FERUT
M T A C No 45 page 30 1954
J N P HUME and B H WORSLEY
TRANSCODE: A system of automatic coding for FERUT
J of ACM Vol 2 No 4 page 243 ,1955
C W ADAMS
Developments in programming research
EJCC page 75 1955
B MONCREIFF
An automatic supervisor for the IBM 702
WJCC page 21 1956
W F BAUER
An integrated computation system for the ERA-1103
J of ACM page 181 1956
J I DERR and R C LUKE
Semi-automatic allocation of data storage for PACT I
J of ACM page 299 1956
C J SWIFT
Machine features for a more automatic monitoring system on digital computers
J of ACM page 172 1957
F P BROOKS JR
A program-controlled program interruption system
EJCC p 128 1957
H SCHECHER
Programming for a machine with an extended address calculational mechanism
C of ACM Vol 2 No 6 page 32 1959
C STRACHEY
Time sharing in large fast computers
Information processing p 336 1959
E F CODD E S LOWRY E McDONOUGH and C A SCALZI
Multiprogramming STRETCH: Feasibility considerations
C of ACM Vol 2 No 11 page 13 1959
E M BOEHM and T B STEEL JR
The SHARE 709 system: machine implementation of symbolic programming
J of ACM p 134 1959
O MOCK and C J SWIFT
The SHARE 709 system: programmed input-output buffering
J of ACM page 145 1959
A W HOLT and W J TURANSKI

Man-to-machine communication and automatic code translation
WJCC page 329 1960
G F RYCKMAN
The computer operation language
WJCC page 341 1960
A HOLT
Overall computation control and labelling
C of ACM no 11 page 614 1960
R B SMITH
The BKS system for the Philco 2000
C of ACM no 2 page 104 1961
Anon
ORION
C of ACM No 2 page 110 1961
R J MAHER
Problems of storage allocation in a multiprocessor multiprogrammed system
C of ACM No 10 page 421 1961
A W HOLT
Program organization and record keeping for dynamic storage allocation
C of ACM No 10 page 422 1961
J FOTHERINGHAM
Dynamic storage allocation in the ATLAS computer, including an automatic use of a backing store
C of ACM No 10 page 435 1961
G O COLLINS JR
Experience in automatic storage allocation
C of ACM no 10 page 436 1961
W P HEISING and R A LARNER
A semi-automatic storage allocation system at loading time
C of ACM No 10 page 446 1961
A B SHAFRITZ A E MILLER and K ROSE
Multi-level programming for a real-time system
EJCC page 1 1961
M B SCOTT and R HOFFMAN
The Mercury programming system
EJCC page 47 1961
T KILBURN R B PAYNE and D J HOWARTH
The ATLAS superivosor
EJCC page 279 1961
M N GREENFIELD
FACT segmentation
SJCC page 307 1962
F J CORBATO M MERWIN-DAGGET and R C DALEY
An experimental time sharing system
SJCC page 335 1962
C A BOUMAN
An advanced input-output system for a COBOL compiler
C of ACM no 5 page 273 1962
N LANDIS A MANOS and L R TURNER
Initial experience with an operating multiprogramming system
C of ACM No 5 page 282 1962
P WEGNER
Communication between independently translated blocks
C of ACM no 7 page 376 1962
J P PENNY and T PEARCEY
Use of multiprogramming in the design of a low cost digital computer
C of ACM no 9 page 473 1962
J W WEIL
A heuristic for page turning in a multiprogrammed computer
C of ACM no 9 page 480 1962
R PERKINS and W C McGEE
Programmed control of multi-computer systems
Information processing page 545 1962
J McCARTHY F J CORBATO and M M DAGGETT
The linking segment subprogram language and linking loader
C of ACM no 7 page 391 1963
A J CRITCHLOW
Generalized multiprocessing and multiprogramming systems
FJCC page 107 1963
M E CONWAY
A multiprocessor system design
FJCC page 139 1963
R V BOCK
An interrupt control for the B5000 data processor system
FJCC page 229 1963
L LUKASZEWICZ
Outline of the logical design of the ZAM-41 computer
PGEC no 6 page 609 1963
H KANNER
An automatic loader for subroutine nests
C of ACM no 7 page 416 1964
J B DENNIS
A multiuser computation facility for education and research
C of ACM no 9 page 521 1964
L E S GREEN
Time-sharing in a traffic control program
C of ACM no 11 page 678 1964
S ROSEN
Programming systems and languages—a historical survey
SJCC page 1 1964
J M KELLER E C STRUM and G H YANG
Remote computing—an experimental system part 2: internal design
SJCC page 425 1964
G E PICKERING E G MUTSCHLER and G A ERICKSON
Multicomputer programming for a large scale real-time data processing system
SJCC page 445 1964
B B CLAYTON E K DORFF and R E FAGEN
An operating system and programming systems for the 6600
FJCC part II page 41 1964
H S BRIGHT
A PHILCO multiprocessing system
FJCC part II page 97 1964
G F LEONARD and J R GOODROE
An environment for an operating system
ACM National Conference page E2.3 1964
G P BERGIN
Method of control for re-entrant programs
FJCC page 45 1964
M P COLE P H DORN and C R LEWIS
Operational software in a disk oriented system
FJCC page 351 1964
H A KINSLOW
The time-sharing monitor system
FJCC page 443 1964
Y CHU
Direct execution of programs in floating code by address interpretation
PGEC no 3 page 417 1965
E L GLASER J F COULEUR and G A OLIVER
System design of a computer for time sharing applications
FJCC page 197 1965
V A VYSSOTSKY F J CORBATO and R M GRAHAM
Structure of the multics supervisor
FJCC page 203 1965

J F OSSANNA L E MIKUS and S D DUNTEN
Communications and input-output switching in a multiplex computing system
FJCC page 231 1965
W W LICHTENBERGER and M W PIRTLE
A facility for experimentaton in man-machine interaction
FJCC page 589 1965
J W FORGIE
A time- and memory-sharing executive program for quick-response on-line applications
FJCC page 599 1965
J D McCULLOUGH K H SPEIERMAN and F W ZURCHER
A design for a multiple user multiprocessing system
FJCC page 611 1965
W T COMFORT
A computing system design for user service
FJCC page 619 1965
B W ARDEN B A GALLER T C O'BRIEN and F H WESTERVELT
Program and addressing structure in a time-sharing environment
J of ACM no 1 page 1 1966
J B DENNIS and E C VAN HORN
Programming semantics for multiprogrammed computations
C of ACM no 3 page 143 1966
G F LEONARD and J R GOODROE
More on extensible machines
C of ACM no 3 page 183 1966
J M WILLIAMS
Design of the real-time executive of the UNIVAC 418 system
ACM National Conference page 401 1966
M J MENDELSON and A WENGLAND
The SDS SIGMA 7: A real-time time-sharing computer
FJCC page 51 1966
R J GOUNTANIS and N L VISS
A method of processor selection for interrupt handling in a multiprocessor system
Proceedings of IEEE no 12 page 1812 1966

Systems recovery from main frame errors

by R. ARMSTRONG, H. CONRAD, P. FERRAIOLO, and P. WEBB
Burroughs Corporation
Paoli, Pennsylvania

INTRODUCTION

"If you look at Automata which have been built by men or exist in nature, you will very frequently notice that their structure is controlled only partly by rigorous requirements and is controlled to a much larger extent by the manner in which they might fail and by the (more or less effective) precautionary measures which have been taken against their failure. There can be no question of eliminating failures or of completely paralyzing the effects of failures. *All we can try to do is to arrange an automaton so that in the vast majority of failures, it can continue to operate.*"[1]

The operational system built by Burroughs Corporation is a communications automaton. It uses conventional computers, conventional computer peripherals, equipment for tying these computers into a communications network, and software for the operation, scheduling, and automatic recovery from failures within both the hardware and software systems.

The significant contribution of this combined hardware and software system is that it provides automatic operation and automatic recovery from errors without duplicating its operations. New electronic telephone systems, for example, provide automatic recovery by duplicating operations and comparing results. This new communications automaton developed by Burroughs Corporation may provide automatic recovery by having only one spare module of each major black box of the system; automatic recovery is achieved by software manipulation of the operating hardware judiciously substituting these spares under program control and keeping track of the performance of each black box operating in conjunction with the other black boxes in the system. Black box modules eliminated from the operating system by the software are repaired off-line by the servicemen.

The reason for developing this new automaton is to provide an ultra-reliable system which is required to give service twenty-four hours a day, seven days a week. The applications for such systems are typified by command and control systems with a much wider sphere of applications possible in the future.

The system designed to meet these stringent requirements is a modular one comprising carefully integrated hardware and software. A sufficient number of modules of each type, computer, memory, I/O controls and peripherals are selected to perform the required function. This configuration is termed the operational system. Then a second system is composed of at least one module of each critical element. A critical element is one whose loss would disable the system. This second system is called the back-up system.

Whenever possible, the two systems run in parallel under the supervision of the automatic recovery program. The operational system performs all required functions and monitors the back-up system. The back-up system constantly repeats a series of diagnostic tests on the computer, memory and other modules available to it and monitors the operational system. These tests are designed to maintain a high level of confidence in these modules so that should a respective counterpart in the operational system fail, the back-up unit can be safely substituted. The back-up system also has the capability of receiving instructions to perform tests on any of its elements and to execute these tests while continuing to monitor the operational system to confirm that the operational system has not hung up.

When the system is running in this duplex mode, failures can occur in either of the two subsystems. The occurrence of a failure in either of the two subsystems initiates the error recovery cycle.

During normal operation, the operating subsystem is running under the control of the operating and scheduling software, and the backup subsystem is operating under the control of the diagnostic software. When a failure is detected anywhere in either system, software control of all hardware is turned over to the diagnostics, and operations are suspended.

This phase of activity is called the error recovery cycle.

The error recovery cycle consists of five phases: error detection, error isolation, reconfiguration, data recovery and resumption of normal operations.

Error detection

The error detection phase relies on the following capabilities and procedures.

Each program is capable of detecting failures in it own equipment.

Each of the program systems has the capability of signalling the other system in the event a failure is detected in the critical modules under its control. The program system receiving the communication will initiate the error isolation phase of the recovery cycle.

Each program system maintains a real-time count in a memory module and periodically updates this count, which indicates the operating status of the system. Each program system periodically checks the real-time count of the other system; this check is made to verify that the count has been updated. If the real-time count is equal to a previously checked value, the error isolation phase of the diagnostic program is initiated. This feature is meant to guard against the few cases in which a computer comes to an unintended stop.

Also, if either system encounters a no-access interrupt while attempting to read from the other system's hardware status table as maintained by the software, the error isolation phase of the diagnostic program is initiated. A no-access interrupt is created whenever access to any system black box cannot be achieved.

In summary, the error isolation phase of the recovery cycle is initiated by either the active system or the back-up system for any of three reasons: (a) the receiving of a communication from the other system to the effect that a failure has been detected in a main frame module of the data processing set, (b) the detecting of a failure in the real-time count of the other system and (c) the receiving of a no-access interrupt while attempting to access one of the other system's memory modules. When the error isolation phase is initiated, the computer that has detected the error is halted, and the *other* computer is used to run error isolation. Thus the computer and memory module normally used to perform error isolation are ones in which confidence has been reasonably established ahead of time. Exception to this rule will exist if the active system attempts to run error isolation on itself in the unlikely event that the back-up system is inoperative due to a previous failure.

A further exception, which also is in the spirit of John von Newman's finite limit to failure protection, is the case when a failure in one machine is of such a nature that it points to the other machine as having failed, when in reality the other machine is good. Such a failure has not been detected in our extended operating experience with this system; if it occurred, the system would go down.

Error isolation

When the error isolation mode is initiated, the diagnostic program becomes owner of all available equipment, and the diagnostic test programs required for locating a fault are executed.

Since all information in the operating system is suspect, once an error is discovered, the diagnostic programs initially destroy all information and load themselves from the one, or preferably replicated duplicate sources on which the programs are stored.

In the checking of the hardware modules, all available magnetic drum storage units are used. Specifically, the test programs executed during error isolation are the computer, switching interlock, memory, and I/O test programs. Following the execution of error isolation, one of the main frame modules will always be updated so that its status is bad. If no errors are detected during the error isolation cycle, the halted computer which detected the error, is assumed to be at fault and is indicated as the faulty module. If an error is detected, an error message identifying the faulty module is constructed and printed on the high speed printer.

Reconfiguration phase

The reconfiguration phase of the error recovery cycle is not executed at a singe point in the cycle but rather at several discrete points. The basis of reconfiguration is the system status table. During the error isolation phase, this table is adjusted to reflect the status of each of the elements based upon the results of the test run at that time. If a module is faulty and marked so in the status table, no attempt to use it is made and the rest of the recovery cycle is relatively easy. If a memory is faulty, the reload program must take this into account and at least some of the program system must be reloaded into areas different from those used formerly.

Data recovery phase

The last act performed in the error isolation phase causes the operational executive program and the data

recovery program and certain fixed tables to be reloaded into the system. In the normal error recovery case, the primary function of the data recovery program is to restore the variable information that existed in memory just before the error which caused the recovery cycle occurred.

The final action of the data recovery phase consists in loading the rest of the operational program system over itself causing itself to disappear slowly and in piecemeal fashion, somewhat like the Cheshire Cat, and normal processing to resume.

Resumption of normal operations

The last function of the data recovery phase causes normal operation to resume.

CRITIQUE

For our original design, the time required to execute the recovery phase just described is in the order of two to four minutes contingent upon the type of failure and the traffic load at the time of failure. During this period of time service is disrupted.

A critical evaluation of that original system revealed that significant improvement could be made by the addition of some particular error detection circuitry. In the original system it was frequently impossible to determine the precise module that failed without running extensive tests using different combinations of equipment. If computer A ran well with all memory modules but one, it was not necessarily true that that memory module was at fault. It was necessary to repeat the test with computer B and the suspect memory. The effect was the diagnostic program required the use of the total system to isolate a failure in a single element.

The addition of some error detection and test circuitry (specifically adding memory parity checking) has made it possible to isolate failed modules in a matter of milliseconds where previously seconds were required. In addition, in many instances it is no longer required to interrupt the operational system to perform the isolation and recovery process.

Further analysis has revealed that, were we to redesign the system or develop a similar system at this time, it would be possible to provide isolation of failed modules in a matter of microseconds in ninety-five percent of the cases where seconds are otherwise required without incurring great expense.

To accomplish this it is necessary merely to recognize as a legitimate goal that individual modules should be readily recognized as faulty when they are, and to provide the requisite facilities to make this determination.

In summary, the significance of what we have done is creation of an automaton which operates and recovers itself automatically. A different system concept is used which saves duplicating the entire set of operating hardware by substituting software controlled testing and interchanging of the few spare modules.

The significant information we have learned that is of value for future software and hardware system design, is that adequate hardware checking within each black box module is necessary such that the source of the errors it indicates are unambiguously reported to the software. We needed more clarity in differentiating between errors occurring in the I/O controls from errors occurring in the I/O peripherals themselves.

The significant accomplishment is that we reached our goal: "that in the vast majority of failures... (the system) can continue to operate."

REFERENCES

1 J VON NEUMAN
The theory of self-producing Automata
1966

Language directed computer design

by W. M. McKEEMAN
Stanford University
Stanford, California

An imaginary absurdity

It is an accident that digital computers are organized like desk calculators—with somewhat worse luck we might have taken the Turing machine as our model. And someone would have been unenlightened enough to prove that, under certain (actually untrue) assumptions, it made no difference. All general purpose machines can compute the same functions, given sufficient time.

Now, if one of the users of an automatic digital Turing machine were to suggest revolutionary additions such as a large random access memory and a special command to add the contents of two memory cells, we could expect to find him attacked on various counts. A design engineer would complain that the additions were *ad hoc* and destroyed the essential simplicity of the Turing machine. Besides they would make the machine ten times more expensive to manufacture. Another user would wistfully agree that the additions were clever and nice but he couldn't afford the expense of reprogramming his entire library of Turing machine programs; a working group would publish a list of accepted standards that were in jeopardy. And finally it would be pointed out that next year's Turing machine would be twice as fast.

The disgruntled user would, of course, take refuge in higher languages. He would use a compiler that would, when it saw the symbol '+', generate the necessary 175 Turing machine instructions required to add two adjacent bit patterns on his tape. He might, when desperation clouded his judgment, attend user's group meetings to make demands for extensions to the compiler that the manufacturer could not reasonably implement on a Turing machine. The following year the user would switch manufacturers and the manufacturer would fire ten programmers for assumed incompetency.

As time went on, the user might console himself with progress: the discovery of a new 167 instruction add routine; hyper-Turing tapes where the bits are recorded in frames 9 bits wide; the appearance of a pipeline micro-parallel Turing machine which, under special circumstances, could execute 27 simultaneous Turing machine operations; and finally, the Ultimate—two time-shared Turing machines working on the same tape.

The solution of some problems would still be beyond reach and the federal government would allocate funds for a really parallel effort—100 Turing machines arranged in a 10-by-10 array sharing 20 tapes on a grid, 10 across the rows and 10 down the columns. Since the machine could be shown to be potentially 100 times as fast for some problems, the best programmers in the world would stand in line to use it, thereby insuring its success by contemporary standards.

The intuitive feeling of our user (that the Turing machine model is not very good) is probably correct, but he must be quite careful in his attempts to demonstrate the feeling as a fact. Given the same level of technology, it is not obvious that the Turing machine does computation less economically. If, for instance, the active parts of the tapes are kept in high speed memory to avoid physical movement and the detectable possibilities for parallelism are fully exploited, automatic digital Turing machines might appear competitive with contemporary computers for a large class of problems. Nor can he claim that his programs are hard to write since the compiler takes care of those difficulties. Furthermore, small evolutionary additions to the Turing machine structure probably won't help much if we take into account all the costs of adding them.

The most obvious point of attack is that Turing machine code is highly redundant—it takes a great many more bits of code to represent a program than the information content demands. Thus the execution of large programs will be affected by the necessity of going to secondary store (or some other uneconomic means) for the instructions. Now compilers and

operating systems are large and frequently used, so they may be used as examples—everybody will agree that it is worthwhile to make them smaller and easier to write.

If the design engineers began to seek more efficient encodings for commonly used sequences of instructions (for instance, direct addressing instead of a sequence of tape moving commands), progress toward the modern computer would have begun.

Our challenge

Today, on contemporary machines, we observe that compilers and operating systems are getting less reliable, consuming more memory, taking more time and systems programmers to develop, use a larger percentage of machine execution time and produce substantially worse results. Keeping our Turing machine analogy in mind, it is interesting to examine the reasons for the trouble.

Since compilers and operating systems are just large programs, we are inclined to simplify the writing job by recourse to a systematic approach in a higher level implementation language. The fact is that most (but *not* all)* such attempts have failed because the resulting programs took too much memory. Since hand coded efforts have succeeded, it is fair to say that the compilers generally were not adequate for the job. Hand coding achieves compactness as a result of a multitude of careful decisions by programmers based on global knowledge of the functioning of the program. When those decisions are in error, we lose reliability. When they depend upon the functioning of some other code, we make program maintenance hazardous. In any case, it takes time to make the decisions and we see the effect in the increased cost of producing the programs. Even for the hand coded programs, the limiting property is size. A great deal of extra effort is spent in making large codes fragmentable, so that execution can proceed in stages with access to secondary store in between.

But it is absurd to expect carefully engineered, very fast, automatic desk calculators to be very good for implementing operating systems or compilers. We are forced to manufacture the operations we want out of sequences of hardware operations with the obvious result that the programs become large. When engineers begin to seek more efficient encodings for commonly used sequences of instructions, progress toward the modern computer may begin.

*Among the successful efforts we include all Burroughs B5500 compilers and operating systems, which are written in variants of Algol-60.

Our objective is to have the machines do our bidding reliably and with the minimum of effort on our part in telling them how. The most obvious, and most obviously profitable step is to make the machines reasonably amenable hosts for the operating systems, compilers and popular languages presently in use.

The palliative

The single most efficacious action that the computer manufacturers can take is to require that the key design engineers have personally implemented at least a production compiler and operating system for the previous machine. By "personally" we mean program design, coding, bit pushing and debugging clear through to a marketable programming system. Nothing less will do. As usual, a little knowledge is dangerous so it would even be valuable to have had him do some application programs. It is very discouraging to hear an engineer "prove" the correctness of a design by analyzing its performance in irrelevant circumstances. It would be somewhat worse to have a machine with some nice operation which cannot be used because it fails in a common but unsuspected circumstance. Careful design will see a total system simulation and completed operating programs before a single circuit is integrated.

The effects of this design approach should be immediate and salutary. It is rather easy to predict some of the results, direction and speed of the resulting designs.

Multiple arithmetic formats

First to go would be the multiplicity of arithmetic formats and the attendant complexities of conversions for every pair of types. We would also demand a consistent set of arithmetic operations that require identical register setups and leave their results in the same place. For example, a divide operation that leaves both remainder and quotient is occasionally convenient, but if that is the only divide that is available, it complicates our register allocation scheme during translation of arithmetic expressions. Only after an engineer has solved this problem in *his* compiler will he be able to say whether few types and consistent operations are worth an extra burden on the hardware.

A measure of the impact of this multiplicity of formats is seen by comparing the actual machine code required for a comparable FORTRAN compiler written on two different stack oriented computers. The stack oriented machine that provided the "programming flexibility" of many arithmetic and register

formats accessible to the programmer required *three* times as much code to implement this part of its compiler as did the machine with the clean, simple and consistent format accessible only through its almost pure stack structure.

A second measure of the programming consequences resulting from a machine design that utilizes many register sizes and allows many arithmetic formats within these registers is a comparison of the percentage of the total compiler required to manage and utilize this "flexibility." Consider a FORTRAN compiler, this time evaluated for a machine that has 4 different register lengths and 3 different arithmetic modes possible in each of these registers. The management, assignment, and manipulation of the 144 combinations of register conditions resulting from this "programming flexibility" required thirty percent of the entire compiler! The complexity required to handle this kind of problem in typical computers is a significant source of errors in code produced by the compiler.

As an aid to the hardware designers who are attempting to design their hardware to optimize software implementation, there are 4 major aspects of arithmetic and register symmetry that are needed:

(1) Different types or sizes of registers must be avoided.

(2) Multiple copies of the same register should not have to be addressed explicity.

(3) Multiple different uses of one kind of register should be avoided. For example, a typical base register oriented machine uses any one of a large set of registers for integer arithmetic, or for base addressing, or for subscript indexing, or for branch addressing. The management of these functions and their allocation among the many possible register locations is a big programming task, a big program always in memory, and it can be a very significant inefficiency at run time.

(4) If different arithmetic formats are used, the hardware should be capable of detecting and manipulating these formats so that the programmer and program need not take into account the different formats. "Typed Data" is an example of this consideration. Where more than one arithmetic format is desirable, we may find a few bits in each word which can be used to distinguish the type of the data. A single arithmetic operation could then be used to trigger not only the correct arithmetic algorithm but also automatic type conversions if necessary. Type bits can also be used to distinguish program from data as well as for many other operations where we want the hardware to be sensitive to the form of the data. The same mechanism can be used to handle access to subscripted variables, indirect addressing, parameters, and read and write protected memory.

Structured addressing

Next to disappear from commercial computers would be the single address instruction. A single address instruction that can use any word in memory is simply a license to commit mischief. In implementing a highly structured language such as PL/1 or ALGOL, we are less interested in addressing all of memory than we are in conveniently addressing all, and just, those variables that are presently active. The use of a stack and a set of register displays is well documented in the literature *as a compiling technique* but rarely found in hardware. Similar comments apply to subscripting and bounds checking, program segmentation, arithmetic temporary stores and subroutine linkage. In each case the machine designer can simplify the compilers and *speed the execution* of programs by confidently establishing some addressing conventions and dedicating some hardware to them.

Specifically for the benefit of our hardware designers, it should be noted that there is one remarkably simple addressing structure common to three of the major programming languages in use today. PL/1, FORTRAN, and ALGOL each use a pair of numbers to represent an address. This number pair has as its first digits the lexical (or nesting) level of the occurrence of the declaration of the name; and as its second digits the actual occurrence of the name in that level in the program.

For example, a typical program might be:

```
begin real A;
    begin real B,C;
    end;
    begin real D;
    end;
end.
```

Corresponding to the names above, the address pairs would be:

Name	*Address Pair*
A	1,1
B	2,1
C	2,2
D	2,1

B and D have the same address pair, and one can tell which name is referred by where one is in the program.

This language characteristic should lead the hardware people to reflect the fundamental elegance of modern languages into the hardware. What we need is a hardware system to compute the real hardware address at run time. For example, the lexical level could

point to a register, and the occurrence level could be an increment to that register.

The significance of this approach is that one is not likely to compile bad object code using this kind of structured addressing. As a result, a programmer addresses his own variables and only those that are immediately accessible. He not only cannot address other peoples' programs but he cannot inadvertently over-write even his own programs.

A second addressing structure which has a major impact on the size and thruput of software systems is addressing by pages vs. addressing by program segment. The basic question underlying this aspect of the discussion is what is the natural program segment size? Analyses of scientific programs we have done shows a peak in number of segments used at 60 words per segment. However, this is an ill defined peak lying between 20 and 100 words per segment, and even at that less than half of the total number of segments used lies in this range. The typical system designer has chosen to impose an artificial quantum of program to address by selecting a page size. Several problems in the programming systems immediately arise and should be noted by the hardware people.

First, in selecting this unnatural cleavage plane in a program, the hardware forces the system programmer into considerable overhead manipulation to handle and bridge the arbitrary cleavage that occurs unpredictably in all programs at run time. If the programmer ignores these artificial boundaries, the overhead goes up catastrophically.

Second, if a large page size is chosen compared to the actual segment size that is located in each page at run time, a considerable part of total memory is wasted. It is an accident if the remainder of the contents of that page happen to be needed unless the programmer has carefully arranged it.

Third, if a page size is chosen that is smaller than the actual segment to be run, there is lost time as the executive program handles the larger number of page faults.

The answer is to use a varying page size equal to the actual size of the natural program segment to be run. This approach is addressing by segment.

A comparative measure of the relative value of these two approaches has been estimated based on programs written first for a modern page addressed machine and second for a contemporary segment addressed machine. This comparison shows that for equal system overhead time measured in percent, that the machine using segment addressing uses approximately ⅓ the memory required by the page addressed machine. This represents a gain of factor of three in terms of main memory utilization!

This difference is of major significance in any commercial system, of course; but in practical terms this is a graphic example of the impact of matching the addressing structure used in the hardware to that used in the programming language.

Secondary benefits of matching these addressing structures occur in the areas of program bounds (or limit registers), in handling arrays, and in handling subroutines. As long as a program stays within its own limits in a page addressed machine, this program can destroy itself or its own subroutines. By contrast, in a segment addressed machine, program cannot do this to itself.

To summarize this section on structured addressing, these are but a few of the many ways for the hardware designer to use the language structures which have been invented to simplify writing and execution of programs.

Algorithmic structures

Some programs, the compilers and monitors in particular, may profit from special attention. A compiler-tested engineer might want a hardware scanner for his next effort; a monitor-tested engineer would likely ask for hardware queues and variable length segmentation structures. It may even be feasible for some programs to disappear completely into the hardware.

An example of the kind of hardware needed to substantially improve compiler thruput would be a hardware scanner. A scanner in programmer's nomenclature is a means for recognizing the character boundaries of the next message to be fed the compiler. This next message is called a token, and a token is an identifier, a reserved word, a special character, a number, or a character string. The means that programmers have used to find the next message, or token, is a programmed routine to scan the sequence of characters waiting to be compiled to identify and mark these boundaries. The characters are passed on as a group to the main part of the compiler. Historically, this scanner routine is the slowest part of compilers. Hardware assistance in this area, again based on the way the programming language is constructed, could provide significant reductions to the programming task.

The idea we are describing here is for the hardware designers to study the algorithmic structures of the important programs that will be written for or run on their systems. There are undoubtedly many system and hardware concepts which can be invented to take advantage of their software algorithmic structures.

As an aid to the hardware designers, some references to the relevant literature are:

Randell & Russell, *ALGOL 60 Implementation*
Academic Press, 1964
Marvin Minsky, *Computation*
Prentice Hall, 1967, page 117
Wirth & Weber, *Euler: A Generalization of ALGOL 60*
Comm–ACM, January, 1966

None of these ideas are new. Nor will they be particularly startling to programmers. The amazing thing is the lethargic pace of the industry in building machines that are directly designed to do the tasks they will be assigned. The measure of success will be in total system performance. It is incredible how few computer customers ever actually measure machine performance for the kind of job mix they normally run before they purchase a new machine. Granted that it is a difficult job; granted that we frequently buy hardware that exists only on paper; granted that the decision is frequently made on other grounds; nevertheless customers and manufacturers must begin to measure actual *performance* if we are to make any kind of progress in system design.

A remedy

The obvious reward of the palliative will be increased performance; the real reward will be the widespread realization that we are not at the end of the line and there are more constructive things to do than argue morbidly about standardization of languages and machines we don't fully understand.

Having surmounted the political problem, we can return to a direct attack on the central problem: How to make the machines do our bidding.

We are our own best model for computer organization. It is our work that we want the machine to do and we have some idea of how we go about it. It is apparent that we can greatly improve the performance of our systems by allowing and controlling unboundedly parallel operation. This system goal implies the invention of a class of artificial languages within which that control will be conveniently expressed. We will be better off if we take natural language concepts as our model than those of conventional programming languages; no matter how machine independent we wished our programming languages to be, they all have been overwhelmingly sequential, arithmetic and random access memory oriented.

The obvious attach for programmers and hardware people together is to devise language that reflects what we want to do and how we do it (for instance, in parallel) and machine structures effective in handling that language. Let us call this method "language directed computer design."

An approach to the simulation of a time–sharing system

by NORMAN R. NIELSEN
Stanford University
Stanford, California

INTRODUCTION

The past several years have seen the development of time-sharing capability on a variety of second generation computers.[1,2,3,4,5,6] For the most part, though, these projects should really be termed experimental efforts. Because the systems were providing a broad or general purpose service, it was not possible to take advantage of the many shortcuts employed by the more specific purpose systems such as those for airline reservations.[7] No assumptions could be made about the size, running time, or bug-free condition of the programs to be executed.

Accordingly, the general purpose systems had to be equipped to cope with the full range of time-sharing problems. File and memory protection schemes became crucial. As programs became larger and longer running, memory space became a scarce commodity, necessitating the consideration of swapping programs in and out of high speed memory from secondary storage, relocating programs, and dealing with core fragmentation problems. System monitors, command languages, editors, and conversational compilers (assemblers) had to be developed. System bugs and malfunctions were frequent. These failures often destroyed files or caused other frustrating delays. Main memory and secondary storage limitations were often restrictive. High system overhead and relatively slow swapping rates often entailed poor response times.

Despite such dire conditions, these systems were very well received. The users felt that the ready access to a large machine and the opportunity for economical man-machine interaction more than offset the aforementioned difficulties. Little wonder, then, that there was widespread interest in time-sharing and a belief that this technique would play a large role in future computing.

It was at this point that the first announcements of the new third generation computers were made. There was every indication that this equipment would permit the development of vastly superior time-sharing systems, for the new hardware would enable many of the limitations suffered by previous systems to be surmounted at a reasonable cost. Not only was the circuitry to be more reliable due to the new solid logic or integrated circuit construction, but memories were to be larger, processor speeds to be faster, and random access storage devices to have greater capacities and higher transfer rates. Many of the former special hardware additions were to be standard features.

Further, certain systems were being marketed especially for time-sharing use, and manufacturers were promising to deliver time-sharing software packages. No longer would an installation have to mount a substantial development effort in order to be able to offer time-sharing capability on its own equipment. Not surprisingly, a tremendous interest developed in these new systems.

The nature of the problem

After the initial excitement subsided, some disturbing aspects became apparent. Despite the promise of these proposed systems, there was little information available on performance or operating characteristics. In terms of equipment acquisition and system operation, this posed significant problems for computation center managers. In particular, consider the difficulties faced in trying to determine the effects of hardware configuration, software modification, time-sharing algorithms, and user behavior upon system performance.

The new hardware was to be quite modular in design, so that one configured a system from a set of standard components rather than ordering a standard time-sharing machine. Hence, a variety of performance information was vital. What was the best configuration to serve a particular set of needs? What was an optimal configuration given an equipment acquisition budget of a particular size? What was the minimum or cheapest

system which would perform adequately in a particular environment? What level of performance could be expected from a specific configuration? What would it cost to provide time-sharing service to 100 users? Despite the importance of this line of questioning, answers were for the most part unobtainable.

A similar situation existed with respect to software. Since an installation might wish to modify the manufacturer's software in order to serve a particular environment more effectively (e.g., incorporating new services or features), it was important to be able to determine the effects of these changes. Would system performance be improved or degraded? Would reprogramming to reduce the overhead associated with certain system operations significantly improve performance, or would it only serve to expose some other problem area? Again, answers were not available.

Associated with every time-sharing system is a set of algorithms for secondary storage allocation, job scheduling, etc. One of the empirically determined facts about time-sharing is that nearly every individual has his own ideas as to what constitutes an appropriate algorithm for each of the various functions. Yet, short of intuition (which is often unreliable in complex situations) there were no procedures by which the merit of suggested alternative algorithms could be judged.

Another problem area concerned the user job stream. A system might have certain characteristics which would significantly impact the processing of particular types of work. Turning the situation around, particular jobs might have characteristics which would affect the total performance of a system. This latter case was of particular interest since Scherr's[8] work on the Project MAC time-sharing system had indicated that users would indeed change the characteristics of their jobs in response to poorer service or higher usage charges. Again, appropriate information about the new systems was not available.

The need to simulate

There was thus a need to develop some type of tool for the analysis of these new time-sharing systems, so that the aforementioned problem areas could be investigated and satisfactory information could be provided. One possible approach would be analytical. For example, Scherr,[8] Schrage,[9] and Smith[10] have successfully developed mathematical models of time-sharing systems or scheduling algorithms. These efforts have not, however, exhibited the flexibility that would be necessary in order to investigate the great variety of topics outlined above. When one is pursuing variations of a particular question, an analytical approach all too often leads to a dilemma. If the new problem is faithfully represented, the model becomes insolvable; if a solvable formulation is developed, the model does not adequately address the problem area being investigated.

Another possible approach would be simulation. Appropriately used, this technique can offer the required flexibility and can serve as an effective tool for system analysis. The simulation work of Scherr[8] and Fine and McIsaac[11] is illustrative. It is not surprising, then, that there has been a substantial interest in simulating the behavior of the new time-sharing systems.

However, as is now being realized, it is not a straightforward matter to simulate one of these new systems. Despite the successful application of simulation techniques to older systems, this approach has not been very effective in analyzing the new systems. A primary reason for this failure has been complexity. Very few persons realized the complexity embodied in some of the standard systems in the third generation lines, much less in the time-sharing systems. Evidence now abounds in the form of software delays, reduced capabilities, performance problems, etc.

The effect of this complexity upon attempted simulations has also been striking. Programming often became long and involved; development time and cost climbed while execution efficiency declined. At times it was necessary to make departures from the system being modeled in order to obtain a reasonable execution speed or a feasible implementation with a particular language. On the other hand, the ability of a simulation model to perform the necessary analysis role has adequately been demonstrated. Thus, the major problem lay in coping with the added complexity in an appropriate fashion.

An attempt was made by the author to develop a generalized time-sharing system simulation which would avoid the trap of complexity without at the same time defeating its own purpose. A reasonably successful model was developed, implemented, and subsequently applied in a study of the IBM 360/67 time-sharing system on order by the Stanford Computation Center. The results of that study have been reported elsewhere[12] and will not be repeated here. Suffice it to say that through the use of the simulation model it was possible to pinpoint a number of problem areas arising from the design of the 360/67 time-sharing system. Further experimentation indicated system modifications which would relieve or eliminate many of those problems. Because of the effectiveness of that simulation, it is appropriate to consider the approach which was taken in its design and construction.

What to simulate

Despite is obviousness, the question of what to simulate was an important one. The four problem areas which were discussed previously (hardware configura-

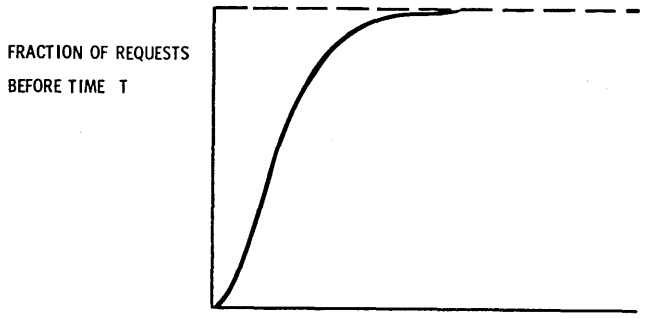
tion, software modification, time-sharing algorithms, and user behavior) were taken to be the goals toward which the simulation should be directed. A careful study of these questions led to the selection of a "software" rather than a "hardware" related level of detail. This permitted attention to be focused at a single level and enabled the resulting model to be much more efficient.

The software operating systems are concerned to a significant degree with the allocation of memory and secondary storage space and with the transmission of data between these two media. Since many of the new systems are paging systems* and since a core image swapping system** is but a special case of a paged system (page size = core size), the page was taken as the basic unit of space allocation. In addition, since the time taken by the system to perform its various functions can normally be measured adequately in tenths of milliseconds, this measure was selected as the basic unit of time.

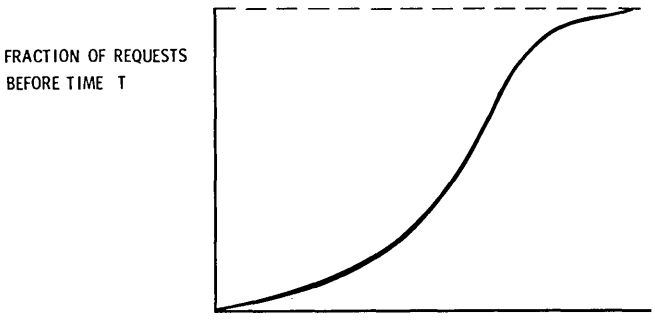
This restriction of the focus of the simulation model to a rather homogeneous level of detail achieved two aims. Not only did it make the model simpler, smaller, and faster, but it also eliminated the need for the much longer simulated time-span which would be required to study the grosser aspects of a system. The effects of these other areas, however, need not be ignored. For example, the more detailed hardware concerns about dynamic address relocation or data channel—CPU memory reference interference can be addressed in a separate simulation or other study concentrating at this level. The results of these other investigations can then be taken into account by inputting an appropriate set of parameters into the main simulation. A similar situation holds for those areas of much grosser detail. Such questions as the long term storage requirements (e.g., from day to day) for program or data files can be handled by a simple pencil and paper study, with the results again being treated as a set of input parameters.

Representation of jobs

The selection of a scheme for representing simulated user jobs can have a significant impact upon the success of a simulation. This is particularly true when such a scheme must be able to reflect behavior realistically in a paging environment. Inasmuch as many of the models of swapping systems use a single distribution to determine the amount of execution time that is to elapse between I/O operations, an obvious approach would be to follow the same line of thinking. A study could be made of the paging behavior of a job and a set of curves developed. Each distribution would indicate the execution time between requests for an additional page given that a certain cumulative execution time had elapsed since the beginning of the time slice or execution period currently allotted to the program. Alternatively, the distribution of execution time between new page requests could be plotted given that a certain percentage of the program was already in memory. Figure 1 illustrates two such distributions for a situation in which paging activity decreases as the percentage of the program already in memory increases.



25% IN MEMORY
Figure 1a



75% IN MEMORY
Figure 1b
Distributions of paging activity

Despite the simplicity of this approach, it has one serious defect. It is possible to determine only that *a* page is required not *which* page is required. Since the particular page cannot be identified, neither can its

*A paging system is one in which memory is allocated in interchangeable sections (pages) to portions of several programs at any given time. Sections of a program being returned to memory for a period of execution will be relocated dynamically into whatever page locations are then available.

**A swapping system is one in which memory is allocated contiguously to one or more programs. A program being returned to memory for a period of execution will always be placed in the same locations that it had previously occupied.

location. Consequently, some assumptions must be made as to the location of each required page—e.g., on a queue in memory waiting to be written out (from the last time slice), on a drum (in any one of a number of rotational positions), on a disk, etc. One way of making these assumptions is to use another distribution which indicates the probability of finding a needed page in each of the possible locations. Note, though, that this is often the very information that is to be obtained from the simulation. The use of such input data makes it impossible to determine the effect of different core management or paging algorithms, thereby rendering the model ineffective for investigating some of the more important problem areas.

A more accurate technique, utilizing the same basic approach, would be to associate with each page a distribution which would indicate the probability of the next reference or transfer being to any particular page. This is illustrated in Figure 2. The master distribution of execution times would then be altered to reflect execution time between references or transfers to a different page rather than to a page which had not been previously used during that time slice. In fact, since detailed information would be kept on each page, it would be preferable to associate a second distribution with each page indicating the possible execution times before a reference or transfer would be made outside that page (see Figure 3).

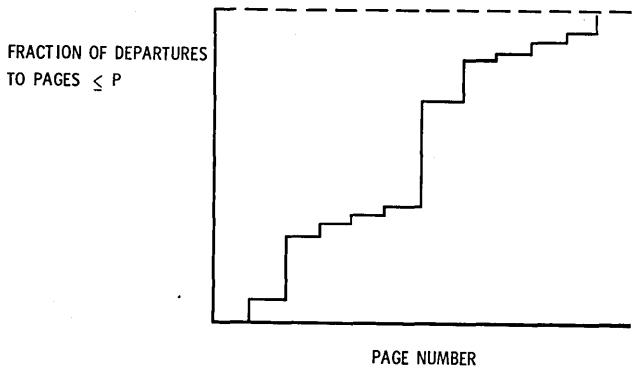


Figure 2—Distribution of page references from page X

Such a scheme, though, is undesirable from both a theoretical and a practical point of view. From a theoretical standpoint a single pair of distributions for each page is simply not able to reflect adequately the behavior of a job. For example, a page might make frequent reference to one set of pages during the initialization phase of a job but to a different set during the result printout. A set of pairs of distributions would thus be required, each pair reflecting the job's behavior at a different stage in its life.

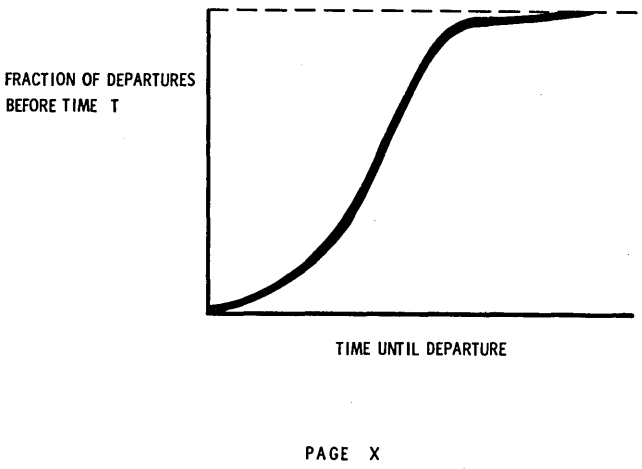


Figure 3—Distribution of execution time for page X

Even if an appropriate rearrangement of the program would enable a single pair of distributions to suffice, the scheme would not be practical for use in an actual simulation study. When one considers the fact that a random number would have to be read or calculated and a table look-up operation performed each time the simulated program referenced or transferred to a different page, the problem becomes apparent. In addition, the storage requirements for the distributions would likely entail very large memory capacities, frequent overlays, or a very compact internal representation (which in turn would increase the number of instructions required for the table look-up).

What was needed was a technique which would reduce the memory and CPU cycle requirement but which would still retain the essential detail about a program's operation. The approach finally selected was a fairly deterministic one. Each stage of a program is represented as a chain or sequence of instructions which describes the specific behavior of that program during that stage. This description indicates a series of specific page references and I/O operations interspersed with execution times. In this way random number generations, table look-up operations, and table storage requirements are substantially reduced.

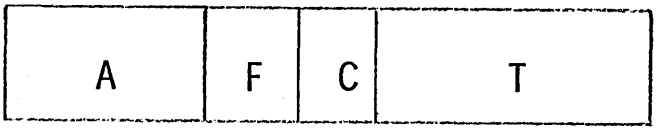
On the other hand, this procedure raises the new problem of storing these detailed job behavior descriptions. This problem can be significantly reduced by permitting the reuse of instruction sequences. That is, if there are repetitive aspects to a job, it is possible to cycle back and reuse the instruction sequence again and again, thereby eliminating the need to duplicate descriptions several times. For example, consider the case of a compilation task. During the syntax checking and table building phase a description of the process-

ing of two statements might be constructed and repeated fifty times in order to reflect the processing of a one hundred statement source file. A description of the code building phase could then be substituted and repeated several times to reflect the execution of the next stage of the compiler, etc. Additional space and execution reduction techniques are discussed at a later point in this section.

The use of a deterministic description involves the loss of a certain amount of authenticity. This is especially true when a particular sequence is repeated many times. However, if the sequences are appropriately constructed, this should not materially affect the accuracy of the simulation. As a further safeguard, the instruction sequences are developed in a two step process.

First, a master or prototype description of each different type of job that might be processed by the simulated system is developed and input to the model as data. Thus, there might be prototypes for compilation, file maintenance, list processing, etc. Then during the actual runs, whenever a job requires another instruction sequence for its next stage, a specific sequence for that particular job will be constructed from the appropriate part of the master prototype. The construction process itself involves a certain amount of randomness, so the procedure is actually semi-deterministic in nature. Thus, each simulated program resembles the appropriate prototype job, but none is a carbon copy of it. Consequently, if several programs of the same type are running on the system, the effects of the determinism are even further obscured. The aforementioned procedures will be more meaningful after a consideration of the way in which job behavior can be described at both the specific and prototype levels.

The instructions which comprise the specific descriptions are each composed of four fields as follows:



A—the page on which the operation is to be performed

F—indicator flags which may affect the operation

C—the op code designating the operation to be performed

T—the raw execution time scheduled to elapse prior to performance of the operation.

A limited amount of experimentation revealed that, by appropriate use of flags, a total of seven different operations would suffice to describe the execution behavior of a job. An eighth operation (C=5) was subsequently added to improve the execution efficiency of the simulation. The function of each of these operations is as follows:

C=0: After executing for time T perform an end of time slice for the current job. This instruction never appears in the job's sequence; rather it is dynamically set up by the simulation as the next operation whenever a time slice end is required.

C=1: After time T obtain page A. That is, if page A is not in memory, set up the necessary queue entries to have it read into memory and place the task in page wait status. If page A is already in memory, return to the execution of the job (for time T of the next instruction). This operation provides the basic means for bringing pages into memory.

C=2: After time T obtain page A as for the previous operation but then increment A by 1. When A refers to the end of a program segment (16 consecutive pages, the last of which is evenly divisible by 16), reduce A by a value which is a function of the flags which are then set in the instruction. This operation can be used to reflect the building of tables and files, since each time the sequence is repeated (up to a limit) the next higher page of the file will be referenced.

C=3: After time T write pages A through A+X out on device Y, where X and Y are specified by the status word representing page A. This operation can also be used to represent the reading of information into parts of pages. That is, the particular pages must be brought into memory before the I/O can begin, since these pages will either contain information to be written out or information that will not be overwritten and hence must be preserved.

C=4: After time T read pages A through A+X (in their entirety) from device Y. This operation differs from the previous one in that any set of available pages can be assigned to receive the input since the contents will be completely overwritten. The previous copies of these pages will then be deleted from their secondary storage locations.

C=5: After time T obtain X consecutive pages beginning with page A, where X is a function of the flags which are set in the instruction. This operation is merely a shortcut to avoid using several of the basic operations (C=1). The pages will be read on different drum revolutions, overhead will be charged as if

there were several separate page fetches, etc. The intent is thus to improve the performance of the simulation without affecting the simulated system.

C=6: Release the job's virtual memory (the logical memory area used by the program as opposed to physical memory) from page A through the end of the program segment. In addition to freeing up virtual memory for reassignment to this or other jobs for different programs, the operation marks the core and/or secondary storage locations of these pages available for use.

C=7: After time T provide a terminal interaction of length A for the job.

With a little practice it becomes quite easy to develop job descriptions in terms of these operations. Table I illustrates part of an instruction sequence describing (in a trivial fashion) the first phase of a compilation. The first instruction indicates that five milliseconds of computation take place using the pages which have already been brought into memory for the job. Then a terminal interaction of fifteen seconds duration occurs. After the terminal input of a new source statement, the first five pages of the compiler are called (pages 37-41). The first two pages of the symbol table (pages 12-13) are next called after four milliseconds of execution. A sixth page of the compiler (page 42) is then called following ten milliseconds of execution. One millisecond later the source statement is placed in the second page of the source file (page 5), and after another millisecond the condensed source statement is placed in the third page of the condensed file (page 22). The last instruction indicates that one millisecond of execution occurs before an entry is made in the control table (page 45).

Despite the ease with which instructions can be developed, it is quite another matter to write compact descriptions which at the same time will be both accurate and efficient. Although a program making use of the semi-deterministic descriptions will execute much faster than one using a multitude of distributions, this still

Table I Sample Instruction Sequence

A	F	C	T	Comment
15	0	7	50	Terminal interaction
37	3	5	0	First 5 pages of compiler
12	0	5	40	First 2 pages of symbol table
42	0	1	100	Sixth page of compiler
5	0	1	10	Second page of source file
22	0	1	10	Third page of condensed source file
45	0	1	10	Control table

does not guarantee that the program will have a reasonable running time. For example, consider the case of a two page program which executes one instruction from the first page, one instruction from the second page, another from the first page, etc. The program itself would not encounter any performance problems during execution, but a simulation having to track each reference would be extremely slow. Hence, a few other helpful features were included in the design of the basic operations.

For instance, in the two page program example given above, it is not necessary to track each transfer between the pages so long as both pages can be retained in the simulated system's memory during the time slice. Accordingly, the shortcut instruction (C=5) could be used to bring in the pages, and a rather long execution time could be specified to take place prior to the performance of the next operation. In this way the paging behavior of the program could be reflected without becoming bogged down at the level of individual instructions.

This does not, however, completely solve the problem. Should the execution time be sufficiently long, one or more time slice ends may occur before the next operation is performed. At the beginning of a new time slice, no operation is performed; rather, execution proceeds using whatever pages are returned to memory by the paging algorithm. In the event that this algorithm does not return all of the pages actually intended to be in memory (in this case two), the shortcut procedure will produce erroneous results. Consequently, provision was made for as many of the previous operations in the sequence as desired to be performed at the beginning of each time slice, so that the actual paging activity could be reflected. The reexecution of these instructions is handled in such a fashion that timing considerations are not distorted.

The ability to remove instructions from a sequence is another efficiency aid. For example, it might be that every repetition of a sequence would be identical with the exception of the first, wherein a read instruction is required to load a program from disk storage. In order to avoid the necessity of constructing a separate sequence specifically to perform the read from disk, a flag is used to indicate that the instruction is to be deleted from the sequence immediately after performance of its specified operation.

The master sequences contain prototypes for all of the specific instructions except time slice end (C=0). At the time that a specific sequence is constructed, the F, C, and T fields of the prototypes are carried over directly to the corresponding fields of the instructions being developed. The A fields which refer to logical pages of the prototype job are translated to reference

the corresponding logical pages of the specific job. The A field of instruction type 7 represents a terminal "think" time, and it is selected according to a probability distribution (see below).

In addition to the prototype instructions, the master sequences contain six different types of control instructions which guide the construction of the specific sequences for the individual tasks. Some of these control instructions can be used very effectively to reduce job description storage requirements; others can be used to introduce some variation into the sequences which are constructed. The format for these additional instructions remains the same, but the A and T fields have quite varied interpretations. The function of each of these control operations is as follows:

C′=0: Set the terminal user headscratch parameter to be A,T. Until this parameter is reset, each terminal interaction instruction that is constructed for a job will be given an interaction time equal to the T value of this instruction plus a random increment drawn from the uniform distribution O,A. This interaction time covers the transmission time for output to the terminal, the headscratch or think time of the programmer, and the typing or input time for the response.

C′=1: Set the loops parameter of the job to be T plus a random increment drawn from the uniform distribution O,A. This parameter specifies how many times the instruction sequence is to be repeated before creation of a new sequence.

C′=2: Terminate construction of the current instruction chain. T indicates the location in the master sequence where construction of the job's next sequence should begin after the now completed one has been executed the appropriate number of times.

C′=3: Transfer the sequence construction process to a new location in the master sequence in the manner indicated by the value of A:

- A=1: Save the index to the current location plus one in the master sequence for a later return; transfer to the instruction at location T.
- A=2: Transfer to the instruction at the location which was previously saved.
- A=3: Store the index to the current location plus one in location T minus one; transfer to location T.
- A=4: Transfer to the location specified in location T.
- A=5: Transfer to location T.
- A=6: Compute a random number R, and compare it with the A fields of the succeeding instructions. When R exceeds A, transfer to the location specified by the associated T field.

C′=4: Set the status word for page A. Depending upon the flags, the status word will be set to indicate that the page should be considered as changed, unchanged, or shared page T during any given time slice. A page is said to be changed if at least one word is stored in it during a time slice, so that the page's secondary storage location (from which it was read) does not contain a correct copy at time slice end. A shared page is one which several jobs make use of in common, so that only one copy of that page need exist in the system at any one time.

C′=5: Reserve or release I/O device, A, which is dedicated for use by a particular job (rather than a particular job type). A flag determines whether an allocation or a release is to be made. In the event that a requested device is not available, a wait (dummy terminal interaction) will be inserted into the instruction sequence and a subsequent attempt made to allocate the device.

The use of the status setting control instruction (C′=4) implies that, for the life of an instruction sequence, any particular page will always be changed (unchanged) during each time slice regardless of when the time slice ends might occur. Although this situation is unlikely to be strictly true in practice, the careful design of the master sequences can make it a reasonable approximation. To try to determine from a distribution at the end of every time slice whether or not each page was changed leads toward the same problems faced previously with regard to the determination of page references from a distribution. Hence, the decision to fix the alterability of pages for the duration of a sequence.

The concurrent existence of several jobs of the same type presents no particular problem with respect to paging, since the logical page numbers of the master sequences are translated to unique job page numbers during the construction of each job's instruction sequences. Device references, however, undergo no such translation, so that every job of the same type will reference the same physical devices. In the case of the direct access devices which are prevalent on the new time-sharing systems, this may be quite realistic. In the case of sequential devices such as tape drives this poses a problem, for in general such devices must be assigned exclusively to a single job.

For this reason the notion of device pools was developed, permitting a number of similar devices such as tape drives to be assigned to a pool covering certain logical numbers. The reserve instruction then causes the assignment of one of these devices to the job, and all subsequent references to this logical unit by that job will be translater to refer to the proper physical unit.

Despite these various shortcuts and aids, great care must still be taken in developing a set of prototype job descriptions. It is, unfortunately, quite easy to develop rather large and inefficient descriptions. On the other hand, quite compact descriptions are possible. For example, the thirty-five job prototypes used in the analysis of Stanford's proposed system required less than 1150 words of 7090 memory. This was not without cost as it took three man-months to put these descriptions together. Nevertheless, given the availability of such a set of descriptions, it is possible to shift job mixes in seconds merely by adjusting the parameters which govern the assignment of jobs to terminals. Further details about the instruction language as well as sample job descriptions may be found in Nielsen.[13]

Output data

Although the development of appropriate performance data for the system being analyzed is a standard problem of simulation, the much greater complexity of the new time-sharing systems has not been without effect. Not only is there a greater variety of possible performance measures, but there is also a vital need to obtain a comprehensive set of measures which might indicate why a particular set of performance figures resulted from a run. That is, given that a system's performance is unsatisfactory during a particular run, some type of information about the conditions underlying those results must be given to the investigator in order that he may develop new ideas for modifications which might improve the situation. Accordingly, it was decided to develop a rather large selection of information during each run. Depending upon the wishes of the investigator, some twenty-five to fifty pages of output can be obtained.

With respect to the simulated work load, distributions can be obtained which indicate the responses received by different job classes (priorities) and by different job types. With respect to the central hardware, statistics can be obtained which indicate CPU utilization (for job execution, overhead, etc.), paging activity, and memory shortages. With respect to secondary storage devices and other I/O units, distributions can be accumulated which indicate the queue sizes and waiting times for both read requests and write requests to each device. Distributions of available space on these devices as well as peripheral equipment utilization statistics can also be obtained.

Further, since it is not possible at the time of development to foresee every piece of information that might later be required from the model, it is necessary to make provision in the design for the satisfaction of unanticipated data demands. The approach selected for this situation was the maintenance of several files into which data could be placed, either on an occurrence basis or on a sampled basis, for later analysis. This mechanism provides a great deal of flexibility at very little cost, particularly since a standard data writing routine enables the collection of new types of information to be readily instituted and the efficient buffering and overlapping of output data to be incorporated. The use of such a routine is also illustrative of the next problem area to be considered.

Modularity of the simulation

The complexity of the new time-sharing systems and the problems already encountered by investigators trying to simulate them tend to make execution efficiency a prominent characteristic of any model. Despite the need for speed, though, it may well be advantageous to compromise this goal to a certain degree. Inevitably it becomes necessary to alter or modify a simulation model after it has been constructed and used. Hence, this factor should be considered during the design phase, so that subsequent changes can be made quickly and easily. More than one model has lost much of its effectiveness when desired modifications could not be implemented within a feasible time span.

Consequently, a fairly modular design was selected for the simulation. To the extent possible each of the major algorithms and functions was isolated from the rest of the model. Thus, for example, the job scheduling algorithm, the data output routine, and the device managment algorithm were each developed as a separate section of the model. Although this produced several obvious execution inefficiencies, the ease with which subsequent changes were incorporated into the simulation clearly demonstrated that the price was very cheap indeed.

Language selection

The choice of a language in which to implement a simulation can often have a significant impact upon the effectiveness of that simulation. Even if implementations were to be based upon a common model, the use of different languages would almost of necessity involve design modifications of varying degrees. In some instances these will be of major import. Five major areas in which language choice is felt are: compilation speed, execution speed, programming difficulty, memory utilization, and compiler availability.

For the case in point, the most important consideration was execution speed. Memory utilization took on a similar importance in view of the model's scope and complexity. Programming difficulty and compilation speed were also important, but only insofar as it was possible to work without undue restrictions. Since the resulting program was to be made available to other investigators, a greater than normal weight was placed upon language availability on various computers.

An assembly language would probably have best met the execution speed goal. However, these languages are machine dependent, and their ease of programming leaves much to be desired. General purpose languages such as Fortran offer a much better compromise. Not only do some compilers have optimizing passes which are capable of emitting quite efficient code, but compilation speeds are often very reasonable in relation to other alternatives. Although some languages do have extensions for simulation work (e.g., Algol-Simula,[14] CORC-CLP[15]), these systems are not in widespread use.

Interpretive simulation languages such as GPSS[16] probably have the fastest "compilation" speeds, but execution efficiency is seriously hampered. A language such as Simscript[17] appears to be more appropriate. Not only does it offer greater flexibility, but the modularity which it imposes upon a program reduces compilation (although not loading) requirements. On the other hand this modularity extracts a price at run time. In contrast to other languages, Simscript offers ways to utilize memory more efficiently. Again, though, a substantial price is extracted in terms of execution efficiency. Some of these problems have been corrected in more recently developed Simscript compilers, but these processors are not as yet in general use.

Accordingly, despite the multitude of simulation languages available and despite the programming advantages of these languages, the simulation was implemented in Fortran IV. This is not to say that the use of Fortran was without a price, for it entailed considerable extra designing, programming, and debugging work. From an effort or convenience point of view, a high level simulation language would have been preferable. Even more general features such as the report generator of Simscript or the automatic data collection of GPSS would have been of assistance. However, these factors were not the primary criteria for language selection in this particular instance.

The fact that Fortran did not have explicit capabilities for simulation was in some ways advantageous. Granted, no timing mechanism is automatically provided and no ready list handling or queueing procedures are available, thereby forcing the programmer to develop his own routines. However, this does afford the opportunity to tailor these routines to fit the needs of the particular simulation, enabling significant gains in program efficiency to be made. For example, the timing mechanism in the time-sharing simulation was designed to operate from two queues in order to take advantage of certain characteristics of the model. The various queues of pages available for assignment or awaiting transfer to or from secondary storage were also treated in specialized fashions. A dynamic memory allocation scheme was employed to supply entries for the multitude of queues contained in the model. A description of the various special design features may be found in Nielsen.[12]

As an indication of the effectiveness of the design and implementation decisions discussed in this and previous sections, the simulation was able to reflect the time-sharing behavior of an IBM 360/67 at an execution ratio ranging between 1 to 2 and 1 to 3 the right way while executing on a 360/50 with a 256K byte memory. That is, one minute of 360/50 time was required to simulate between two and three minutes of 360/67 operation. This was much better than had been anticipated for a simulation at this level.

Parameter inputs

Having designed and implemented a simulation model, the next step was to construct a set of input data for an actual application. The problems associated with the development of appropriate parameter values are similar to the ones faced by investigators who simulated some of the existing time-sharing systems and will not be elaborated upon here. However, the word existing does indicate one pitfall.

For example, at the time that information about the 360/67 time-sharing system was desired, that system did not exist. Consequently, it was necessary to simulate the *intended* rather than the actual design of that system. Further, as on any project of such magnitude, implementation does not always proceed along the design path. Thus, not only must the design and its changes be monitored, but so must the implementation.

CONCLUSION

The simulation of the new large-scale third generation time-sharing systems is not just a move along the continuum of time-sharing simulations. Rather, there has been a rather sharp increase in the complexity of the new systems which has necessitated a correspondingly sharp change in the manner in which they are simulated. The approach illustrated in this report is not the only one nor is it in all likelihood the best one.

Despite the advantages, the use of semi-deterministic job descriptions is not an easy task, and the development of specialized list structures and processing routines is not without hazard. However, this approach has been employed quite successfully in the study of one of the new time-sharing systems, and it may help others to refine their own ideas about simulating these systems if not actually to develop their own simulation model.

REFERENCES

1 F J CORBATO M M DAGGETT R C DALEY
An experimental time-sharing system
Proc 1962 SJCC vol 21 Spartan Books Washington DC pp 335-344

2 T E KURTZ K M LOCHNER JR
Supervisory systems for the Dartmouth time-sharing system
Computers and Automation vol 14 no 10 pp 25-27 55 October 1965

3 J McCARTHY S BOILEN E FREDKIN J C R LICKLIDER
A time-sharing system for a small computer
Proc 1963 SJCC vol 23 pp 51-57

4 J H MORRISSEY
The QUIKTRAN system
Datamation vol 11 no 2 pp 42-46 February 1965

5 J I SCHWARTZ E G COFFMAN C WIESSMAN
A general purpose time-sharing system
Proc 1964 FJCC vol 26 Spartan Books Washington DC pp 397-411

6 J C SHAW
JOSS: A designer's view of an experimental on-line computing system
Proc 1964 FJCC vol 26 Spartan Books Washington DC pp 455-464

7 *A survey of airline reservation systems*
Datamation vol 8 no 6 pp 53-55 June 1962

8 A L SCHERR
An analysis of time-shared computer systems
MAC-TR-18 Massachusetts Institute of Technology Project MAC June 1965

9 L E SCHRAGE
Some queuing models for a time-shared facility
Unpublished doctoral dissertation Cornell University February 1966

10 J L SMITH
An analysis of time-sharing computer systems using Markov models
Proc 1966 SJCC vol 28 Spartan Books Washington DC pp 87-95

11 G H FINE P V McISAAC
Simulation of a time-sharing system
Management Science vol 12 no 6 pp BI80-194 February 1966

12 N R NIELSEN
The simulation of time-sharing systems
Comm of ACM vol 10 no 7 pp 397-412 July 1967

13 N R NIELSEN
The analysis of general purpose computer time-sharing systems
Document 40-10-1 Stanford University Computation Center December 1966

14 O DAHL K NYGAARD
SIMULA—an ALGOL-based simulation language
Comm of ACM vol 9 no 9 pp 671-678 September 1966

15 W E WALKER J J DELFAUSSE
The Cornell list processor
Cornell University 1964

16 IBM CORPORATION
General purpose systems simulator III user's manual
Form H20-0163

17 H M MARKOWITZ B HAUSNER H W KARR
SIMSCRIPT: A simulation programming language
Prentice-Hall 1963

Experiments in software modeling

by D. FOX
Fox Computer Services
New York, New York
and
J. L. KESSLER
IBM Corporation
Poughkeepsie, New York

INTRODUCTION

The objectives of a software support package for any computer system can quite readily be defined by the potential user of such a system. He will quickly point out that he wants the system to provide all the features and functions that he requires and that it should occupy minimal storage space and consume minimal time. Obviously, the designer of a software support system faced with these impossible objectives finds his life to be one of constant decision making as he comes up with the compromises that give the best possible approach to this solution.

In this paper, we will concentrate on techniques to aid in one aspect of that decision making: the effect the inclusion of a given feature has on the performance characteristics (time and space) of the final hardware/software systems. Indeed, what is the effect on performance of particular design alternatives that one might choose in implementing these features?

In the past, these decisions have been made purely on an ad hoc basis. Experienced system programmers have dashed off kernels of the code representing the final implementation, then extrapolated from these what they thought would be the size and speed of the final product. With sequential processors and with software systems implemented by small groups (where each individual understood in detail the operations of the complete software package), such an approach was acceptable. However, with the increasing degree of parallelism allowed by our more recent computer system designs, and with the exploitation of that design by systems programmers, performance prediction has become a problem of almost overwhelming complexity.

Only in the last few years have hardware measuring and monitoring techniques permitted the detailed study of performance characteristics of such systems after they were in operation. However, postponement of the availability of any performance data until a checked-out hardware and software system is ready for measurement can be catastrophic. Too many irrevocable software-design decisions will already have been made. Thus, it seems obvious that some technique for predicting the performance of systems programs and for evaluating the impact of various design alternatives on that performance must be considered.

For the past three years, a group within IBM Programming Systems has been working in cooperation with Fox Computer Services to study and develop techniques that will permit the prediction of performance characteristics of such systems under a variety of hardware environments. In addition to providing preliminary design analysis, such a performance model can permit the study of comparative configurations, aid in the preliminary analysis of proposed input/output devices, and can aid the designer in optimizing performance within a given configuration.

Within IBM, earlier attempts in the direction of systems modeling emphasized the hardware aspects of modeling. In particular, major importance was attached to central processor cycle times, peak transmission rates, and so on. The primary objectives were to determine how much core storage, how much direct-access device capacity, and how many lines, terminals, and channels were going to be needed to do the proposed job. The software, if it was described in any detail at all, was generally represented as a series of queueing algorithms describing how the terminals were to be polled, how the accesses to the mass storage devices were to be queued, and so on. It became apparent, however, that in order to provide a useful software support package, a radical departure from this total hardware orientation must be made.

Requirements

Now, let's consider just what is required of a set of techniques and tools for aiding the software designer in this way. Looking at any of our major systems today, we realize that the effective use of hardware is largely under the control of the software. So, primary importance should be attached to the software model. This is not to be interpreted as a lack of concern for the hardware performance; rather, it is a realization that the efficiency with which the software utilizes the potential of the hardware is a major factor in determining total system performance.

First, it became obvious that we needed an integrated model of the complete hardware and software sytem. We found that, although some subsystem modeling (for example, modeling only a storage-location algorithm or only a disk-queuing algorithm) can give considerable aid to a designer in a specific area, subsystem models taken alone and not evaluated against the design of the total system often give extremely misleading results. Some subsystem modeling is of value, but final decisions can only be made after the incorporation of that subsystem model into the total system model.

Secondly, if the model is to be constructed by (and used by) software designers, its structure should be meaningful to them. That is, the model should appear to have the same logical structure as the software being modeled.

A third requirement is that a model, or collection of models, must be highly parameterized. That is, the designer must be able to change the entire environment or parts of the environment without recompiling or reprocessing the entire model. For example, hardware characteristics and configurations should be readily changeable, as should the description of the layout of data on the modeled devices. But, all this parameterization and flexibility leads immediately towards the requirement for large volumes of data to describe the environment in which the model is to run. To free the user who is not interested in all this flexibility from having to provide such a great amount of data, a method of storing default cases and readily accessible standard descriptions of this environment is required.

Fourth, a very important requirement of such a system is brought about by its intended use as a design tool. If a designer is to gain anything from such a tool, he must have it at his disposal at all times. This implies that the designer is the modeler. In a large operating system environment, where many groups are designing and implementing components, this implies that there are a great many modelers. Their models are interrelated and have communication problems just as their component counterparts in the real system have communication problems. So, the need for modeling techniques capable of combining small modular units with minimal interface problems is evident.

Fifth, if the modeling system is to be used as the design tool and used effectively, it must be run constantly, many times per day. It must be detailed, for to influence design it must closely parallel the logical design of the system. This implies that we must have an extremely detailed model that runs very fast.

The sixth goal is one that developed out of early experiments, when we discovered that the modeling technique must be able to encompass all potential hardware and software systems that a designer might be interested in. Specifically, it must be able to handle multiple central processors (not necessarily homogeneous in performance characteristics), handle multiple memories (not necessarily homogeneous in their characteristics), be accessible from a number of processors, and must be able to simulate multiprogramming activity within the central processors.

The first thing one discovers in looking at a set of results from a simulation effort is that some of the results trigger an interest in statistics that do not appear on the first report. Often, simulation must be redone, or at least rerun, to produce the results that are now of interest, and, again, the second set of results probably leads to a desire for a third and a fourth, and so on. Therefore, the seventh requirement is that the modeling techniques must allow, first, that such additional data can be acquired and, ideally, that it be acquired with a minimum of model rerun time.

Design criteria

The approach to modeling a total hardware/software configuration for the purpose of design support and evaluation must differ from previous simulation efforts in several important respects. It is software-oriented and makes no simplifying or averaging assumptions about resource demand. Modules of the subject system are reflected as modules in the model, so that the precise effect of module replacement can be calculated by inclusion of the new module in the existing model.

Since we wish to model the total system and, at the same time, reflect each significant module of the subject system, it is clear that the modeling language must emphasize subroutine capability with explicit and easily described interface procedures. This approach eliminates a difficulty that has plagued modeling systems up to now—the requirement that a modeler who is interested in only one part of the system understand and simulate the rest of the system and its environment before he is able to get any results about his particular part. Our techniques permit modular construction of the model, with an emphasis on explicit interface design

and evaluation. System design changes can be isolated and their effects evaluated without changing any other parts of the model.

In order to study the behavior of the software models under a variety of conditions, one must be able to vary both the hardware and software environment in which the model is executed as well as change the simulated work load. Our techniques provide for an input language to describe these environmental considerations and permit the model easy access to the descriptions.

A word or two about choice of a simulator would seem appropriate at this point.[2] Ours was not the first attempt at total system simulation, and it is important to understand the reasons for the rejection of previously used, more standard simulation techniques and languages. We needed a tool for designers and design evaluators. Therefore, the language must follow as closely as possible the methodology of systems designers and analysts. This is accomplished by isolating software events and bringing to the fore the obvious characteristic of software design—that it is composed of modules of logic, each of which is designed to operate sequentially, but whose interaction may be sequential or parallel, synchronous or asynchronous, consequential or logically independent. Another constraint on our design was an historical rather than a technical one. It was not within the scope of the project to implement a new compiler or translator, nor to modify an existing one in any way. In addition, we were aiming towards a technique which would permit us to perform simulation experiments using System/360.

A study of existing simulation techniques led us to the conclusion that, in terms of scope, capacity and performance, existing simulators would have severely limited the size of the system we could model, the length of model runs which could be achieved economically, and the level of detail to which systems could be studied. But the most important characteristic that appeared lacking in existing models was modularity. Our choice, then, was to outfit FORTRAN with mechanisms for implementing simulation models via CALLs, because it offered a most flexible program and subroutine linking structure that was known and understood almost universally among designers.

Evolution (experiments)

Now, let's consider how the need for specific techniques evolved from some of our experiments with software models. One of the earliest was the modeling of the IBM 7010 operating system, using a GPSS III[3,4] This was merely a first attempt to answer two basic questions:

1. Can Programming Systems applications be modeled successfully?
2. What are the problems involved with simulating systems programs, and how would one define a good simulation system to solve this problem?

The result of this experiment was an indication that such techniques could indeed provide promising results. However, the difficulties involved in producing the right kind of model were great, and the process was a very expensive one.* It was evident that much more experimentation and exploration had to be done before we really understood the software modeling problem.

Some modeling was also done in a special-purpose computer simulation language,[1] which proved to be easier to use but suffered from many of the same difficulties found in the GPSS approach. Following this, some of the earlier designs for Operating System/360 language processors, Assembler, COBOL, and FORTRAN, were modeled using a GPSS derivative. These early models did produce reasonable performance predictions. The developers, however, felt that the assistance they were getting from the model was not sufficient to warrant the effort to maintain it.

As these models were being developed toward the end of 1964, Robert Ruthrauff of IBM Programming Systems began to experiment with performance-prediction models that emphasized to a great extent the queuing techniques employed by the input/output supervisor of OS/360. He coded these early performance models in FORTRAN, choosing it primarily because of the familiarity of the programming community with the FORTRAN language. Parallel to this effort, Fox Computer Services had been engaged in producing a performance model for IBM of the IBM 7090 FORTRAN compiler, to be used in evaluating design alternatives that were available to implement that processor. It soon became evident that the fundamental approach was identical, and that much could be gained by combining these efforts in an attempt to develop the more comprehensive set of techniques that were needed to solve the software modeling problem.

By this time, Mr. Ruthrauff's I/O model had undergone a series of six revisions. As an experimental

* Some of the reasons cited in the summary of this project appear below:
1. A model of sufficient detail to be useful was too large and too slow.
2. Techniques of model writing required to represent certain hardware software events and activities in these languages were disagreeable to software designers. (The logic of the model did not "look like" the logic of the modeled program.) Also output traces of events did not follow software events.
3. No dynamic input capability.
4. No model overlay capability.

vehicle, Fox Computer Services and IBM combined to produce another model as an attempt to describe the design of the OS/360 operating in the Primary Control Program environment. Once again, the modeling was done, not by the designers, but by an outside group. Once again, the design of the model did not closely approximate the design of the system. However, a number of things were learned:

1. That a model could be created that would provide valuable performance data to the designer.*
2. That the input/output activity could be closely approximated, and
3. That the particular implementation did not provide enough flexibility for modifying either the hardware or software environment in which various pieces of the model were running.

Thus, in the next two experiments, the OS/360 model was upgraded and further techniques developed. By early 1966, the last of this series of models was completed. The model included a limited multitasking capability, SPOOLING, priority scheduling, and the FORTRAN E and H and COBOL F processors of OS/360. At the close of the OS/360 modeling experiment, two studies were made:

1. To discover the level of accuracy to which such a model could be calibrated.
2. To determine how effective we had been in parameterizing the models. In other words, how much variation in the behavior of the modeled system and in the environment in which those models were executed could be made by merely changing control card values, and how valuable were the results that could be obtained by making such minor variations and reruns?

The validation and calibration experiment proved to be a great success. A set of simulated job streams was run on a series of three System/360 configurations embodying different central processors and different input/output configurations. The results were measured with hardware monitoring devices. The same set of simulated job streams was passed against the model. By varying the input parameters governing the times various portions of the model consumed, the deviation of the model from the actual execution of the operating system was less than 5% in any area studied. The model was validated to the level of overall processor time, phase times and major input/output requests. This included the fetching of program phases, the opening and closing of data files, and all input/output requests, and accounting for both device time and central processor time necessary to initiate the input/output operation. For any one configuration, the results could be readily calibrated to yield zero deviation. The variation caused by moving from one configuration to another could be traced to certain processor phases where assumptions about the consistency of instruction mixes across all system programs was obviously incorrect.

The second study, although educational, did not indicate as much success. We found that very little variation in the modeled system's behavior could be made by varying control-card parameters alone. Almost any experiment of interest required a change in the code of at least one of the models.

By this time, we felt that we were beginning to understand what techniques would be required to allow designers of software systems (and components of those systems) to use modeling effectively as a tool for predicting the impact of design alternatives on their final product. From this point, we set off to develop a set of subprograms that would provide the software modeler with the type of functions our experiments indicated were required. It is not a new simulation language or even a simulation system. The user is free to write his own simulation system using the set of subroutines to perform simulation-unique (and software-simulation-unique) functions. While this does place some extra burden on the modeler, he is not hampered by any built-in restrictions or omitted features.

Evolution (techniques)

Having chosen FORTRAN, then, our job was to outfit the software modeler with "CALL"-able mechanisms for the construction of models which would emphasize the events and methodology of total hardware/software system analysis. Again, it must be made quite clear that modification of the FORTRAN language or its compiler was not permitted.

The first mechanism developed was called the Clockworks, whose function was to queue models in time, keep track of events, and pass control to the appropriate submodel when its event time became due. In its simplest form, the Clockworks would accept requests for synchronous delays (in which the requesting model would be suspended until the delay was effected) and asynchronous delays (in which the calling model could continue without suspension while the scheduled delay, when its time completed, could cause the activation of a second model, in parallel with the first).

The scope of the systems to be studied with these techniques includes the entire gamut of hardware/software systems under past, present, or future development. This means that the event-scheduling mechanism must be able to handle multiple CPU's, multiple and

* This model was actually used in support of some design changes which were later incorporated into the OS/360 FORTRAN H compiler. It showed the performance advantages of a certain input buffering/blocking scheme and predicted the advantages of the multiple compile capability.

parallel tasks, flexible hardware configuration descriptions, resource management, task scheduling, and task dispatching. Consequently, the Clockworks mechanism must be open-ended as to the number of tasks to be executed in parallel and the number of future events to be scheduled. This flexibility is achieved by using list processing techniques and dynamic storage allocation for internal Clockworks operation.

This description of the requirements on the time-handling mechanism does not differ markedly from similar mechanisms available in other modeling systems. There are, however, at least two features which are peculiar to hardware/software system modeling. The first is that time itself may not be absolute. That is, a given task may request a resource for a fixed amount of time, but the actual amount of resource time required to satisfy the request may have to be increased because the resource's effective performance is degraded by shared utilization. In particular, a program designed to complete in 100 microseconds on a give CPU-memory combination may actually consume 125 microseconds because concurrent channel activity is "stealing" 20% of the cycle times for data fetch and store activities, which are not related to or known to the requesting program.

Another requirement on this Clockworks is the ability of a single subprogram to represent a model that may be operating at the same instant of simulated time on more than one set of data, in different modeled environments, and in different states of completion. This requirement, known as the reentrability attribute of the model, led us to design a set of mechanisms called the Model Transfer Mechanism, to help the Clockworks relieve the user of the necessity of including logic to test the current state of the system or his model.

Whenever the Clockworks transfers control to a model, it sees to it that all status information and registers are updated in accordance with the particular usage of that model. Thus, for example, at the same instant of simulated time, a model of a compiler could be processing the second input card of deck A on CPU 1, and the same model could be in the middle of processing the tenth input card of deck B on CPU 2. Whenever a request for time on CPU 1 completes, the model is given control in such a way that it simply continues, without any logic to test in which usage it is involved (Figure 1). This is because the CPU number, as well as the pointers to deck A and card 2, are placed in the same storage locations where CPU number 2, deck B, and card 10, respectively, will be placed upon completion of a time request on the second CPU. That is, each usage of the model effectively operates upon the identical variables, which differ only in their values.

The basic unit of data around which the Model

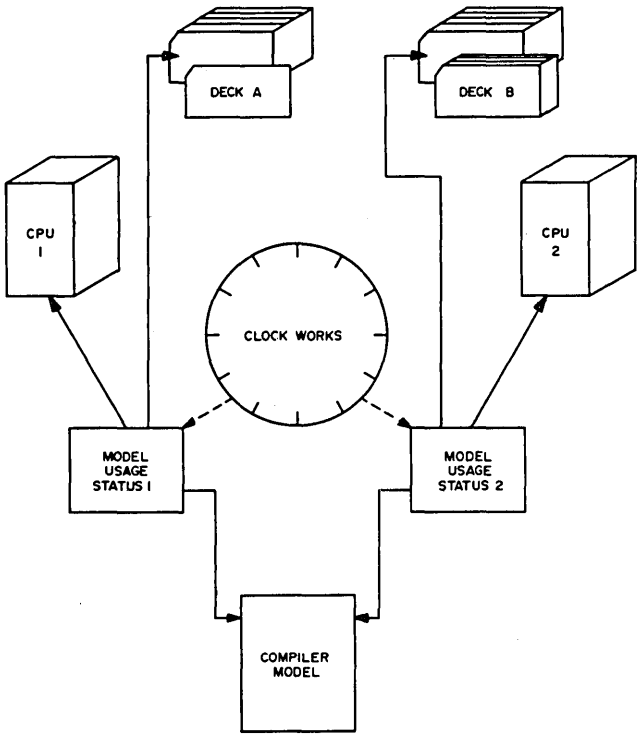


Figure 1—Relationship of clockworks to model usage

Transfer Mechanism operates is the Model Block, or MB. There is one MB for each model usage. The MB serves as a repository for all data and status information required to restore control to a model after its operation has been interrupted or temporarily discontinued. Among the information stored in the MB or, in the case of variable-length information, pointed to by the MB, are:

1. All System/360 general registers, including the address of the next instruction in the component model, usually written in FORTRAN. This is the return point for model resumption;
2. All intermediate or temporary storage currently being used by the model;
3. Call arguments or input parameters to the model; and
4. Return information for model completion.

Every model references data and is controlled only through its Model Block. This permits the Clockworks (via the Model Transfer Mechanism) to restore control to a model by pointing to its currently operative MB. All data (such as card number and deck number in our earlier example) are automatically referenced correctly because the MB is the focal point for such reference.

Until now, we have been using such terms as "task," "model," and "CPU" as if their meaning in a model

were self-evident. Actually, this is far from the case. Our approach implies a specific interpretation of these terms, and it is important to understand these meanings.

It was pointed out earlier that software design consists of modules of sequential logic, each of which can operate as a unit in parallel with others. The term "task" denotes such a unit. It is actually a handle (called the Task Block, or TB) on which is hung descriptions of model operations and events that operate sequentially. Thus, the task is the largest unit of sequential logic in this approach (See Figure 2).

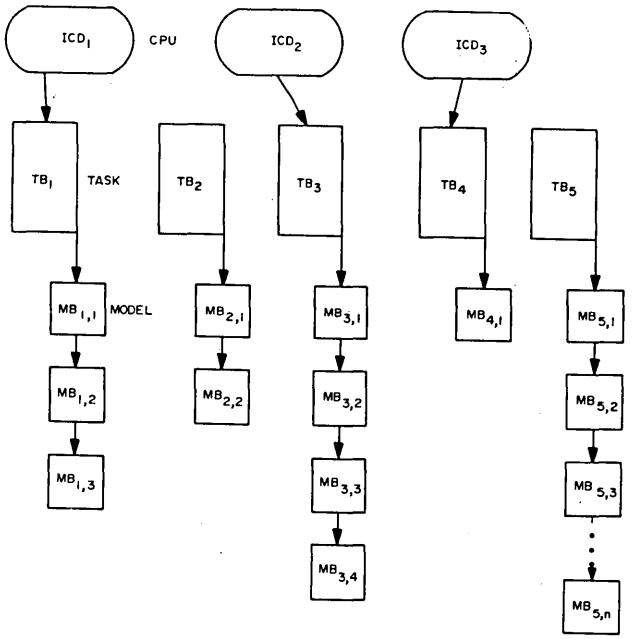


Figure 2—Independent control domains and related tasks

The CPU, on the other hand, is a special case of a more general modeling concept, the Independent Control Domain, or ICD. An ICD is an environment in which tasks operate. It has two properties: first, each ICD can contain a task in operation, regardless of task operations in any other ICD; second, no more than one task can operate in an ICD at any one time. A CPU is an ICD insofar as task seizure of a Central Processing Unit implies that no other task can use that Central Processing Unit at the same time. On the other hand, a multiprocessing system permits one task to operate in each CPU concurrently. (Of course, Central Processing Units that do permit multiple, simultaneous instruction sequences would be modeled as multiple ICDs.)

Another kind of ICD that can arise in systems where operator or user interaction is important in evaluation is the operator domain. This ICD can be a conceptualization of the human part of the system, which presumably will be occupied with just one task at a time.

A "model" is a smaller unit of sequential processing, usually a single subroutine that corresponds to a load module in the subject system. Tasks usually consist of several models invoked either sequentially or in push-down fashion, with the model invoked for the task returning to its predecessor upon completion. Each active task has exactly one model currently operative at any time; it is called the "top" model for the task.

Several other mechanisms are available to the software/hardware system modeler, but we will only be able to touch upon them briefly. The first is dynamic storage allocation, called the WORK mechanism. This consists of a single array of COMMON storage named WORK. Both users and mechanisms can get and free WORK blocks as needed. Addresses and pointers used by, and passed between, models are always understood to be subscripts of the WORK array. We have written a general List Processing Mechanism that is used by both our modeling subroutines themselves and by their users. This mechanism constructs lists in WORK.

Finally, we have provided generic input and output capabilities that allow the user to define multiple input streams, to request statistical as well as snapshot and trace output, and to read and write table or data overflow onto direct access data sets.

To illustrate the use of some of these mechanisms, let us follow the action of a typical model statement. Suppose that the top model of a task is a compiler that wishes to invoke a supervisor function, such as opening a file. The compiler model would

CALL SVC ('OPEN ', arguments to OPEN)

Control is passed to the OPEN model by calling the model transfer service "SVC" to mimic the System/360 supervisor call interruption. Since OPEN itself is a model, it would operate under an MB. Prior to the OPEN CALL, the MB list of this task is simply as shown in Figure 3.

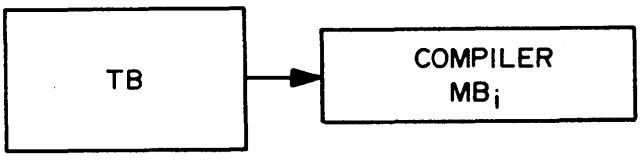


Figure 3—MB list prior to OPEN CALL

After the call, the processor MB is pushed down and OFTEN becomes the top model as in Figure 4. Notice that for as long as the OPEN model is in use, it constitutes the operation of this task, and its events are the events of the task. As soons a it issues a

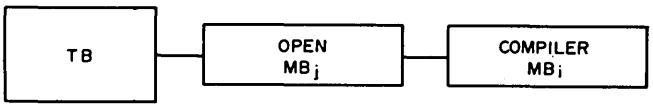


Figure 4—MB list following OPEN CALL

RETURN

the list is restored as shown in Figure 5 and the processor model resumes at the model statement just beyond the call for OPEN (this location having been saved in registers in the MB). Any data or intermediate variables used by the processor are attached to its MB and, therefore, will be undisturbed by OPEN's operation.

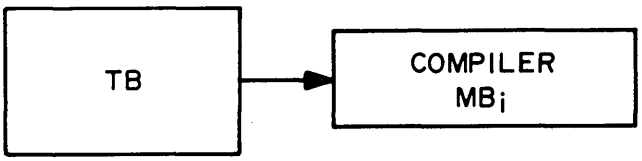


Figure 5—MB list following RETURN from OPEN

Structure of a simulation

The structure of any system can generally be described as consisting of input/output processing components. Simulation is no different. In particular, for ease of implementation and use, our software modeling package is divided into three processors: the setup or preprocessor, the execution processor, and the postprocessor.

The preprocessor digests the data which describe: the hardware device characteristic and hardware configuration, the structure of data on the external storage devices, the simulated job streams* and, finally, the parameterizations of the models themselves.

Since we allow considerable flexibility in the definition of this environment, the amount of data needed as input is very large. To free the user of this software modeling system from providing all this data, standard definitions and default cases are cataloged and put at his disposal. For instance, complete hardware configurations are cataloged so that he can identify them with only one card and invoke all the data defining the devices and the connections of the devices in that configuration. Similarly, standard jobstreams, data set layouts, and model descriptions are also cataloged. The preprocessor extracts this environmental information from libraries as ordered by the user and sets up tables and temporary data sets that communicate this information to the rest of the simulation. In addition, the preprocessor accesses a large library of models, selecting those that describe a specified operating system environment, and adds or modifies models as requested by the user. The output of this processor is a load module, ready for execution, and a set of input data ready to drive the created model.

The execution processor is made up of all the software modeling mechanisms, plus models selected from the library and models introduced by the user for this particular simulation run. The output of this phase is a standard statistical summary, plus a log of simulation events. The standard statistical summary presents, first of all, the amount of time that the central processor was controlled by each of the various tasks, as well as other information about the status of each task. In addition, statistics are maintained on the utilization of all data sets, channels, control units, and devices, and the amount of contention for these entities. For direct-access devices, the amount of time lost due to rotational delays and positioning is also provided. Finally, model usage is summarized. Such data as the amount of time that each model was in control of the modeled system and the number of times each model was used, are recorded. Various levels of logging are available to the user at execution time; indeed, within his own models, he may elect to record an event on the log at any time he chooses.

The postprocessor is really a report generator for processing this event log in a variety of ways. The user may elect to get a complete flow trace of any or all models through the postprocessor. He may be interested in summary or subsummary totals accumulated about selected events on the log. The postprocessor permits generation of many types of reports and, in addition, includes the capability of adding user routines at a number of exit points so that the report capability may be extended by the user in a specialized way, if he so desires.

Summary of techniques

This has been a brief look at some of the important features of an approach in which the total model consists of small modular components. The features are recapitulated here:

1. Total hardware/software modeling with emphasis on software language, events, and characteristics.
2. Explicit modules with explicit interfaces following the structure of the system as closely as possible.
3. Flexible, modeler-defined input/output facilities.
4. Dynamic storage allocation and reallocation.
5. List processing.

* Description of the work load to be handled by the modeled system.

6. Time processing, including dynamic degradation reflecting hardware performance degradations.
7. Wide scope of subject systems.
8. Mechanism to permit reentrant modeling with standard subprograms.

Status

Most of the techniques and mechanisms we have described have been implemented. We certainly do not believe that we have solved all technical problems, or perhaps even scratched the surface. However, the authors are convinced that a tool of this nature is desirable to control performance characteristics during the software design process.

REFERENCES

1 P H SEAMAN
On teleprocessing system design Part IV the role of digital simulation
IBM Systems Journal 5 No 3 175-189 1966

2 D TEICHROW J F LUBIN
Computer simulation-discussion of the technique & comparison of language
Communications of the ACM Vol 9 No 10 723-741 1966

3 R E EFFRON G GORDON
A general purpose digital simulator and examples of its application
IBM Systems Journal 3 No 1 22-34 1964

4 H HERSCOVITCH T H SCHNEIDER
GPSS III—an expanded general purpose simulator
IBM Systems Journal 4 No 3 174-183 1965

Design, thru simulation, of a multiple-access information system*

by LOUIS R. GLINKA,
R. MICHAEL BRUSH** and ALAN J. UNGAR
IBM Corporation,
Gaithersburg, Maryland

INTRODUCTION

The popular concept of a multiple-access processing system as a common facility for many users (at remote terminals) with varied processing requirements, has over the past few years received considerable attention.[1-3] The research for proper hardware and software design solutions has been primarily directed at installations which would support a large and diverse group of users with varied and sometimes complex applications, thus implying the requirement for a powerful computer and large memory storage.[4,5] The usefulness of these time-sharing systems stems from their capability to provide computing power to any user when, where and in the amount needed.

Within this broad classification, there is a type of system which can be characterized by its dedication to support a limited number of applications, and a relatively small number of users whose characteristics are considerably more uniform. This special purpose type of time-sharing system entails the simultaneous usage of multiple terminals on-line with a central information processor by a homogeneous group of system users performing the functions of information storage and retrieval. One example of such an installation is in the military, where non-data processing oriented operations personnel operate and maintain a central depository of information in a field-based tactical intelligence operation.

The characteristics of this system, which is considered to be typical of its kind, imply certain design requirements. The task of formulating an appropriate system design in response to these requirements is greatly facilitated by a modeling and simulation process. The broad objectives of this activity were to represent in a model the characteristics of, and potential designs for, such an information handling system, and to evaluate the system trade-offs between data processing equipment (as characterized by certain parameters), program design and system performance.

This paper discusses the previously cited military system in terms of the job functions to be performed, the man-machine interaction, and the implied requirements of an appropriate hardware and software design It further defines the measures of system performance, the representation of the system, and the presentation of quantitative results. Finally certain conclusions are drawn which center on the effect of the central processor's main memory size and hence the processing strategies as constrained by available memory capacity, on system performance.

The system

The operational role of this system is to operate and maintain a central depository of information in support of an adjoining environment through the maintenance, retrieval and dissemination of data.

Data concerning a broad spectrum of subjects is received and handled in the system in varied forms and from many sources. In general, the system operators (i.e., users) categorize this data by subject and add it to existing collections of similar information. There is a requirement of the system to allow a large number of different data sets; i.e., dissimilar in structure and content.

The integrity and currency of the system data base is maintained by the users through either the adding of data, or the deleting or modifying of existing data. These update actions may affect one or more entries in one or more data sets. In addition to creating and

*This work was supported in part by the Air Force Systems Command, Aeronautical Systems Division, Wright-Patterson AFB, under Contract No. F33(657)-67-C-0158.
**Currently with Electronic Data Systems Federal Corporation, Washington, D.C.

maintaining the data base, the operators use this collection of stored information by effecting retrievals of selected data. These usually are in the form of either highly selective (i.e., discriminating) inquiries, or requests for summarizations of large amounts of stored data. In any case, the user generally requires the output presentation in a particular format on a selected device.

Thus the system is dedicated to supporting a constrained set of applications (viz., information storage and retrieval functions) while on the other hand requiring considerable latitude and generality in the type and amount of data to be handled, as well as in the flexibility of data retrievals and forms of output data presentation. This implies the requirement for a generalized data handling program solution which is independent of the actual structure and contents of any data set in the system.

The system users interact directly with the system rather than through some data processing service group, and usually are operations personnel who are skilled in areas other than data processing. Thus there is the requirement that the operator training requirements and the likelihood of operator errors in system use be minimized. A hypothesized design responsive to both requirements is a sequential process of problem input whereby each operator action results in processor acknowledgment of the action and a processor request to the operator for additional inputs, or processor output of the requested data. This bilateral man-machine dialogue continues through a sequence of information exchanges upon which decisions are made and available options are exercised until the desired end result is achieved. The system user is in effect compiling a job statement by stepping through a pre-determined network of problem-definition nodes. This scheme entails a hierarchical structure in which the user chooses a problem definition path, as a function of job requirements and personal experience or proficiency in system usage, and furthermore is able to discard all non-appropriate steps. At each node (or step) as depicted in Figure 1, the user responds with an appropriate information input according to an instruction cue presented by the processor. The problem definition network has the added attribute of facilitating subsequent modifications and additions and can be arbitrarily large and complex.

The simultaneous usage of multiple terminals online with the system processor constitutes an environment in which each terminal's request for processing is competing with all other requests for use of the processor and other system resources. Furthermore, there is a requirement for rapid access to and response from the system such that at each terminal, the system will appear to be completely captivated by that user. Thus a system supervisor program is required which will schedule the work to be done in the system and the use of system resources according to some priority scheme.

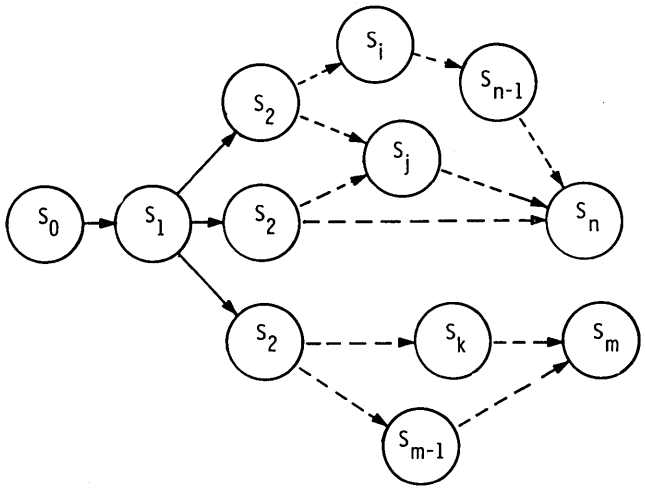


Figure 1—Job definition network

Since only a small fraction of the total information (including the data base) contained in the system can reside permanently in core storage there is a requirement that the system control program provide for the overlapping of input/output operations with other processing, within the constraints of available core space.

In the total time period of turning around a job, the interaction between man and machine varies such that during the job definition phase the user is active, while during the job processing phase the user is passive or in the background. Similarly the processing requirements vary during this total time period, in that the servicing of job definition steps through a pre-established network entails considerably less processing than does the execution of the compiled job statement. Thus there is the requirement for multiprogramming support within the system control program.

Due to the restricted nature of the job types (i.e., data storage and retrieval) to be processed, and the availability of a generalized data handling program to process any of these jobs, there exists then a certain amount of predictability and commonality in the system operation similar to that normally associated with batch processing applications. In this regard, the processing of a job once initiated to completion appears *a priori* to offer advantages in terms of system thruput performance. This can be rationalized on the basis of the reduction in overhead time achieved

with overlapped program swaps and stems from the predictability of processing requirements. A further extension of this theme towards batch processing operations is based on the commonality of processing requirements among such similar jobs and thus a consideration of the merits in grouping jobs for execution through the processing sequence. This type of processing strategy is termed here as "multi-job" processing.

The servicing of job definition steps through the system control program requires application programs which maintain the status (or position in the job definition sequence) of each terminal, interpret (and/or translate) user inputs, and determine and provide for the subsequent instruction (cue) to the operator.

The previously stated requirement for a generalized data handling program implies the existence in the system of a program to cover the possible eventualities of job statement inputs. However partitioned into modules, this application (or problem) program is, in effect, a group of common routines. In order to provide the required flexibility and adaptability in their intended usage these routines must be serially reusable (if not re-entrant) and relocatable; i.e., following the execution of a routine relative to one job, the same (main memory) copy of that routine can be used again in regard to another job, and furthermore the routine is capable of execution in any physical part of the main memory. Thus there is a requirement that each active job in process have dedicated work space.

Figure 2 depicts a grouping of data processing equipment units which is considered representative of the hardware requirements of systems of this type. The use of specific parameter values to characterize a particular equipment configuration for the cited military system is treated in a subsequent section.

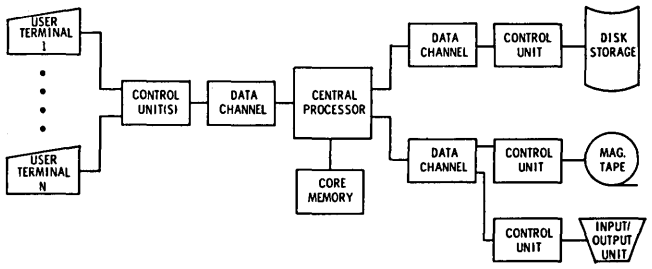


Figure 2 — Equipment configuration

Performance measures

From the system user's point of view, there are at least two important quantitative measures of system performance; viz., job turn-around-time and job processing time.

The *average job turn-around-time* is defined as,

$$\frac{\sum_{i=1}^{N_c}(T_{j\ out} - T_{j\ start})_i}{N_c}$$

and the *average job processing time* is defined as,

$$\frac{\sum_{i=1}^{N_c}(T_{j\ out} - T_{j\ in})_i}{N_c}$$

where,

- $T_{j\ start}$ is the time at which the job definition sequence began.
- $T_{j\ in}$ is the time at which the last job definition step for the job was completed.
- $T_{j\ out}$ is the time at which the output of the job results to the terminal CRT display is completed.
- N_c is the total number of jobs processed to completion during the run.

The job turn-around-time encompasses the total involvement time of the user during the course of accomplishing one job. The job processing time is the time when the user is, in effect, passive and awaiting final output results.

There is at least one additional measure of system performance which is relevant to the system user and is frequently discussed in regard to the broad class of multiple-access systems; i.e., the average system response time per user action (or input). However for this special purpose type of system, where the processing requirements during the job definition phase and the job processing phase differ significantly, the cumulative result would be biased and hence was not considered a meaningful measure of comparative performance.

In addition to the above measures of system performance, the behavior of internal system queues and equipment component utilization statistics are common indicators for analyzing the operation of a system and evaluating the performance of a design solution.

The model

The system model entails certain operational assumptions which reflect the envisioned operating environment and a postulated software and hardware design. Since the modeling activity constituted just

part of an over-all effort directed at establishing an appropriate design, then by necessity only gross estimates of the functions and dimensions of an envisioned design could be made at this time.

In principle, the desirability of using parameters rather than constants in a modeling activity is well recognized. The use of constants in this specific activity, however, can be rationalized on the basis of its limited initial objectives.

Equipment

The operating characteristics of the equipment components in the model were taken from descriptions of available hardware. While the modeled units are not all of the same manufacturer, they are compatible in the modeled configuration. The following equipment was modeled:

- The central processing unit (cpu) has a 32 bit word length and a 2.5 μ second memory cycle. The processing speed was represented by an instruction execution rate of 100K instruction per second. The core memory options available with this cpu ranged from 16K to 64K words, in increments of 8K words.
- A random access disk storage device, combining the control and four drives in one unit, was configured on a separate high speed data channel. The unit has a total capacity of 200 million bits and a data transfer rate of 720K bits per second. The modeled unit reflects the capability to seek and read (or write) simultaneously on separate drives. The following additional operating characteristics were incorporated into the model: 20 ms average rotational delay time, and mechanical access times distributed over a range of 59 to 272 milliseconds.
- A magnetic tape unit was configured on a separate data channel. The data transfer rate was 16.9K six-bit characters per second and the start/stop time of the unit was modeled at 6 milliseconds.
- Eight (8) operator terminals with self-contained buffers, an output-only plotter, and a printer were configured on one data channel. The operator terminals were assumed to include a CRT console with a 900 character display image and an alphanumeric keyboard. The X-Y plotter had a 150 eight-bit characters per second data acceptance rate and a 200 millisecond start/stop time. The printer was modeled with a 300 eight-bit characters per second data acceptance rate and a 6 millisecond start/stop time.

Data base

The system was assumed to contain 16 data sets, ranging in size from 100 records to 13,900 records. Each data record had a fixed length of 400 characters. The total data base assumed was approximately 15.5 million eight-bit characters.

Data set descriptor tables, which specify in detail the structure and contents of each data set and permit interaction between the data and a generalized data handling program, were included in the modeled design. A primary index and additional indexes, as a function of data set size, were included for each data set.

System jobs

All jobs were entered into the system via the user terminals. The selection of a job at each terminal was done on a random basis. The following types and their percentages of occurrence were modeled.

- single record retrieval (44%)—which resulted in an output to the terminal CRT display.
- multi-record retrieval (22%)—which required the sequential processing of n data records, where n was arbitrarily set at 8, and resulted in an output to the CRT display, printer and X-Y plotter.
- multi-data set retrieval (12%)—which required the sequential retrieval of m data records, one from each of m data sets, where m was arbitrarily set at 2, and resulted in an output to the CRT display and printer.
- single-record updata (22_)—which in addition to the data maintenance processing, resulted in a logging of the record on magnetic tape and a confirmation-of-update message output at the CRT display

All outputs to the CRT display, printer and magnetic tape were assumed to be 900 characters. Output to the plotter was 400 characters.

For single record retrievals and updates, data sets were selected with equal probability from the 16 available sets. For multi-record retrievals the data set was selected with equal probability from the seven largest sets. The identical procedure was used for the selection of the first set in a multi-data set retrieval. For the latter, the second data set was randomly selected from the nine remaining sets. The number of data set indexes to be processed in each job was a function of the job type and the size of the associated data set; in general, the number of indexes processed ranged from two to five.

Man-machine procedure

The formulation of each job statement was achieved through a sequence of man-machine data exchanges. Each job would be defined by an operator in s distinct steps; based on the procedural complexity of the job mix modeled, a constant value of $s = 10$ was considered to be representative of the average job. At each step, the user viewed a pre-recorded procedural message sent to him by the processor relative to the data set(s) of interest, he decided what input was required, and made the input via either terminal control key or input keyboard action. For each step in the job definition sequence the operator response time was assumed to be $t = 4$ seconds. On completion of the last step, input was complete, the terminal was locked-out from further user actions, the input job entered the job queue and was processed in terms of the requested retrieval or update of data.

To provide the maximum load on the modeled system each operator initiated the job definition sequence for a subsequent job immediately upon receiving output of the prior job.

Control program

The system control program was assumed to provide the following functions which were reflected in the model.

- Work scheduling in the system queues such that entries in the job queue were serviced on a first-in first-out basis.
- Interrupt control and handling in the following order: disk, magnetic tape, user terminals, plotter and printer, supervisory functions, application program; such that, an interrupt on I/O-complete would be honored during the execution of an application program.
- Program loader capable of relocating programs during transfer from secondary storage.
- Input/output control which maintains the queue of I/O requests and provides for overlapping of I/O operations with other processing, within the constraints of designated available space.
- Operator-terminal control which sequences the interpretive programs required to validate the operator's job definition input and establish the appropriate subsequent response.
- Multi-programming support of the servicing of job definition steps with the processing of defined jobs. A higher processing priority was assigned to the former so as to minimize the job definition sequence time. Note that in the job definition sequence, subsequent user inputs in compiling a job statement are by definition based on prior outputs to the user.

  Multi-job processing support which establishes the group of defined jobs to be batch processed through the required processing sequence. The size and contents of the group was dependent on the designated available work space. In the model, multi-job processing was defined such that, the selection of the Q oldest jobs in the job queue, to be processed (program by program) to completion in a batch, was performed so as to,

$$\text{minimize } [S - \sum_{i=1}^{Q} J_i]$$

  where S is the work space available for processing these jobs, and J is the work space required to process the Q_i^{th} job.

Application programs

The model reflects the assumed design of a generalized set of data storage and retrieval programs for the processing of all system jobs. The envisioned program was partitioned according to the following functions:

- (P1) job examiner—selects job from job queue and routes to appropriate checking routine.
- (P2) initial job processing—checks statement structure and coded elements.
- (P3) table build—compiles job statement using data set descriptor tables.
- (P4) macro-generation—creates macro instructions from tables built by P3.
- (P5) job execution—executes macros built by P4.
- (P6) retrieval processing—fetches data from disk by a direct access method.
- (P7) update processing—creates and modifies data records, stores records on disk unit, and updates indexes.
- (P8) output message processing—formats the resultant data for output presentation to the appropriate device(s).

The processing programs were assumed to range in size from 500 instructions to 4.9K instructions; with a total of 13K instructions for all component processing programs. The basis for this characterization of the processing programs is largely empirical in nature. Similarly the amount and nature of executed code in each program module was established in light of the system job types. The representation of processing times was established on the basis of these factors as applied to the modeled processor's operating characteristics.

Core storage allocation

The allocation of available core space was based on the following assumptions of space requirements.

- control program 11.5K words
- buffer for 8 operator terminals 4 K
- application programs (maximum) 13 K
  (minimum requirement—4.9K)
- work space per active job 3 K

Since the postulated control program was not sufficiently defined to allow a dimensional partitioning, it was assumed to reside at all times in 11.5K words of core. Collecting other assumed minimum requirements for core space resulted in the gross determination that 24K words of core would be the minimum size modeled. In pursuit of the core size vs. system performance question, three core sizes were modeled: 24K, 32K, and 40K words. For each core size, used to the maximum extent possible considering the size of each of the job processing programs, the number of programs which could simultaneously reside in core storage was established. As depicted in Figure 3, three basic designs resulted; viz.,

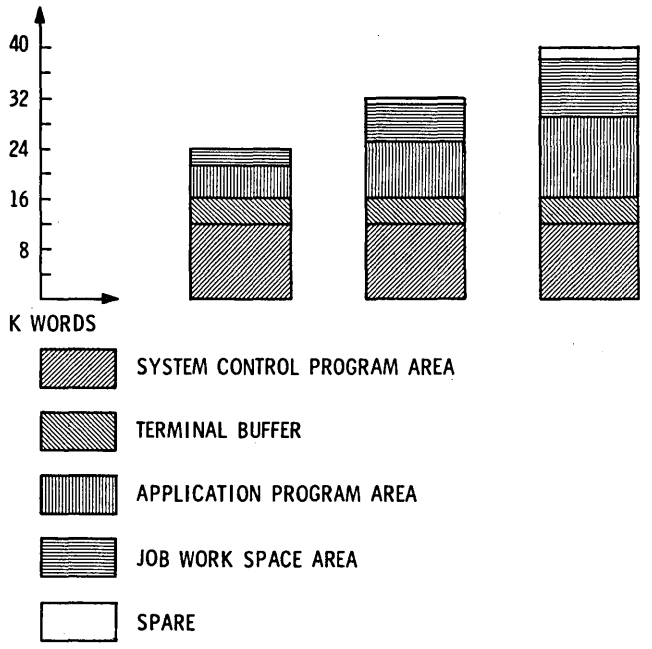


Figure 3—Designated core memory space

- 24K core size—a predominantly program overlay type of design, and allowed for only one job to be processed at a time.
- 32K core size—a design featuring program-read overlap with processing, and provided for the grouping of two jobs through the processing sequence; i.e., multi-job processing limit of two.
- 40K core size—an essentially resident type of design, with a multi-job processing limit of three.

Thus the increases in available core space, in 8K word increments, were reflected in the model by providing in each case an increased capability in processing.

Disk storage allocation

The allocation of the disk storage space available on the four mounted disk-pack drives was performed for the following categories of information: programs, pre-formatted output messages (i.e., instruction cues for the user), data set descriptor tables, and data sets. As depicted in Figure 4, information (e.g., processing programs, data set descriptor tables, etc.) which was to be requested with a predictably high frequency was stored near the median cylinder of each disk drive; thus attempting to minimize the arm motion incurred in disk accesses.

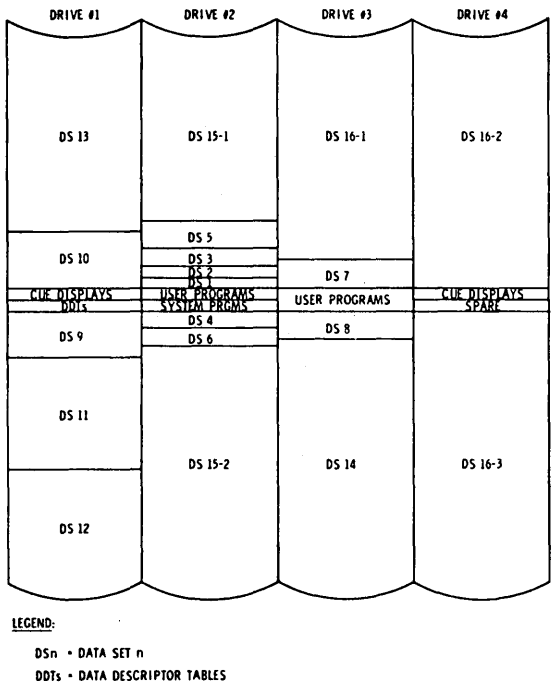


Figure 4—Disk storage map

Operation of the model

The operation of the model is depicted in gross form in Figure 5.

To provide a common basis for comparison of run statistics, the following conditions were maintained for all runs:

- simulated system operation time was arbitrarily set at 15 minutes;
- the system was initialized by setting the status of the processing programs as being initially in or out of core, according to the core size design;
- the disk arms were prepositioned to cylinder #50;
- all user terminals were activated simultaneously at the start of each run.

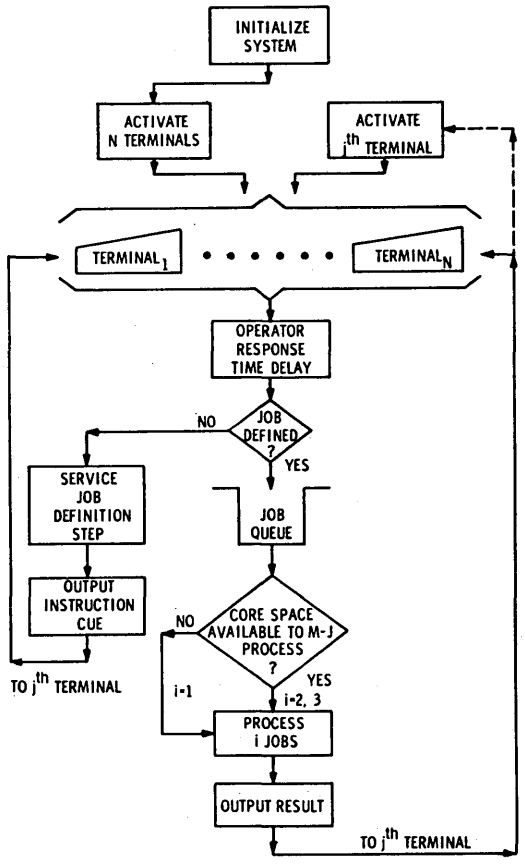


Figure 5 — Operation of the model

Results

The results presented here were selected as the most pertinent of the large volume of statistics resulting from the simulation. They are partitioned into four categories, viz., job queue statistics, job turn-around-time, job processing time, and equipment utilization.

Job queue statistics

The average number of jobs in the job queue and the average time spent by a job in the queue are shown in Table I for five different simulation runs. Results are included for the multi-job processing scheme as described previously (ref., The Model), and furthermore for a single job processing scheme at the 32K and 40K word core sizes. The latter two runs were achieved by inhibiting the multi-job processing capability; i.e., setting the multi-job limit equal to one.*

	MULTI-JOB PROCESSING			SINGLE-JOB PROCESSING	
CORE MEMORY SIZE (WORDS)	24K	32K	40K	32K	40K
AVERAGE NUMBER JOBS IN QUEUE	.12	.09	.08	.11	.11
AVERAGE TIME IN QUEUE (SECONDS)	.50	.34	.29	48	44

Table I — Job queue statistics

The results illustrate some advantage in the multi-job processing scheme. Specifically, in the 32K system with multi-job processing, jobs spent 28% less time in the job queue than for the 32K system with a single-job processing strategy. The 40K multi-job processing scheme showed about a 30% advantage over its single-job processing counterpart. From the user's (macroscopic) view, these differences could be considered operationally insignificant.

That the average number of jobs in the job queue was less than one job for all sizes suggests that the processor, even for the 24K design, frequently found the queue empty and had to wait for work. Clearly the rate of input was lagging behind the processing rate.

Job turn-around time

The average and maximum job turn-around times for the 24K-40K multi-job processing runs are depicted in Figure 6. Job turn-around time as defined earlier, is the time measured from the start of the job definition sequence to the time of the final job output.

As shown in Figure 6, there were no operationally discernible differences, again from the user's view, in average job turn-around time for the three core sizes; the values are essentially constant at 42 seconds.

The maximum job turn-around times were encountered over relatively few jobs at the beginning of each run as a result of the simultaneous terminal starts. Thereafter, the majority of the jobs fell very close to the average value. Specifically, for the 24K design,

*In this case, the 40K word design required 31.5K words of memory storage.

93% of all the jobs were turned-around in <44 seconds; for the 32K and 40K designs, 95% and 96% were turned-around in <44 seconds, respectively. For each run there were essentially 150 jobs processed in the system. The sharp drop in the curve from 24K to 32K shows again the effect of multi-job processing when queueing is a factor.

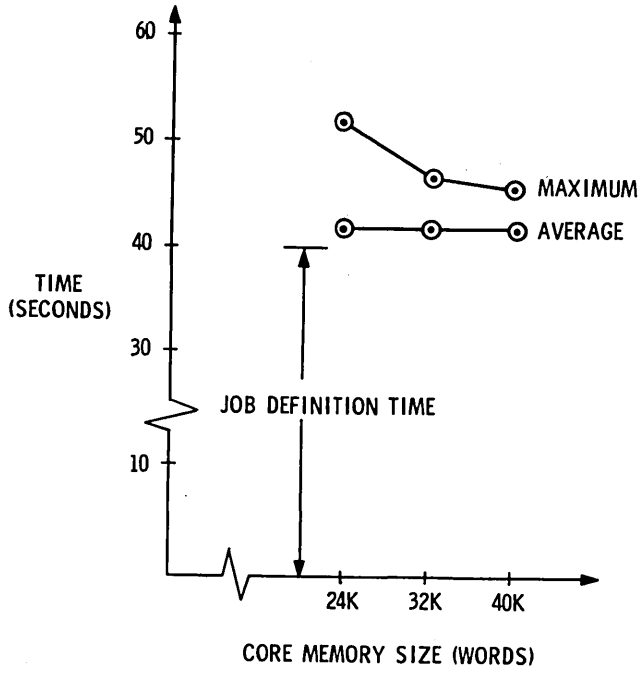


Figure 6—Job turn-around time (multi-job processing)

Recalling that 40 seconds of the average job turn-around time is attributable to user response delays, one can see in Figure 6 that the average rate of processing defined jobs is on the order of two seconds. Here, then, is the explanation of the lack of buildup in the job queue. With each user requiring about 40 seconds to define a job, the combined group of users can input an absolute maximum of one job every five or six seconds, while the processor can handle defined jobs at about three times that rate.

Job processing time

There were differences, though small ones, in the average processing times for each run. Processing time, in review, is defined from the time of input of a defined job at the job queue to final output at the terminal. Figure 7 depicts the average processing time for multi- and single-job processing, as well as for a defined-"effective" multi-job processing time.

Curve A of Figure 7 is drawn from the results for the multi-job processing scheme. In the 32K model, two jobs were taken from the job queue whenever possible and processed together to completion. In

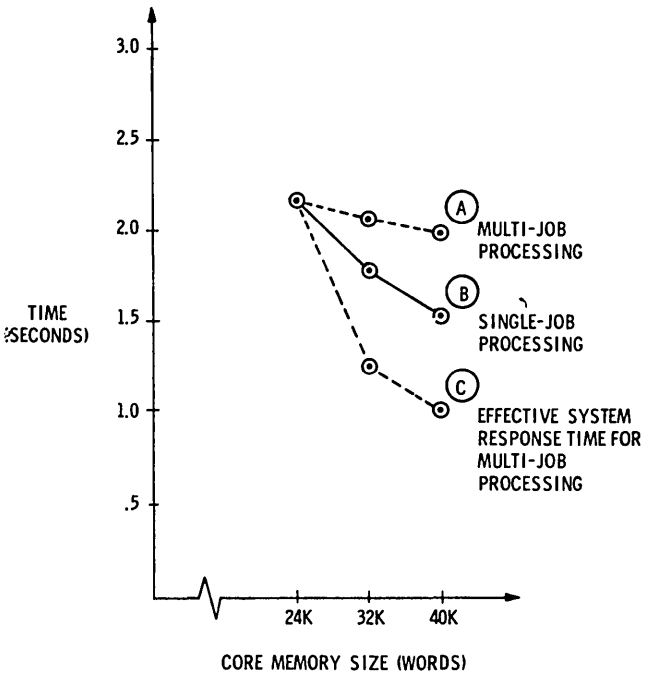


Figure 7—Average job processing time

40K, up to three jobs were processed concurrently when they were available in the job queue. This approach moves more jobs through the system in a given amount of time than does the single-job processing scheme, by avoiding, among other overhead functions, repetitious program reading. For example, if two jobs processed individually take two seconds each, the two processed together might take a total of 3.9 seconds for a time savings of 0.1 second. Note that for the jobs processed individually, the average is two seconds, while the two processed together, by virtue of waiting for each other, average 3.9 seconds each in the processing area. On occasion, the user might in fact sense this real delay as his job, the first in the job queue, waits for other jobs in the batch to be completed. Nevertheless, the system would experience an over-all improvement in output rate. Curve A shows the average job processing time, perhaps more accurately called average time in the processing phase, as seen by the terminal user.

Curve C represents an alternate viewpoint of the multi-job processing capability by showing the "effective" system response time. The effective system response time for a job in a multi-job processing system is defined here as the quotient of the processing time for a batch of jobs processed together divided by the number of jobs in the batch. Thus, from our prior example, two jobs processed together in 3.9 seconds would be said to have an effective system response time of 1.95 seconds each. But this view of system

performance although useful to the system designer is more relevant to the system manager than the system user.

Since there were no significant buildups in the job queue, the multi-job processing solution had little opportunity to produce any important processing gains relative to the single-job processing scheme. A comparison of Curve B, drawn from the results of the single-job processing runs (multi-job processing inhibited in 32K and 40K) with Curve A demonstrates that this was in fact the case.

Equipment statistics

The cpu utilization, disk channel, disk arm and magnetic tape channel utilization statistics for each run are listed in Table II.

	MULTI-JOB PROCESSING			SINGLE-JOB PROCESSING	
CORE MEMORY SIZE (WORDS)	24K	32K	40K	32K	40K
CPU UTILIZATION (%)	9.0	6.6	5.8	9.0	8.9
DISK CHANNEL UTILIZATION (%)	24.2	19.8	11.9	24.2	12.3
DISK ARM UTILIZATION (%)					
• ARM #1	6.9	7.5	5.6	8.9	5.7
• ARM #2	18.4	11.3	3.8	15.4	4.4
• ARM #3	11.1	10.2	10.1	11.4	11.2
• ARM #4	2.5	3.2	2.7	2.7	2.6
MAGNETIC TAPE CHANNEL UTILIZATION (%)	.2	.2	.2	.2	.2

Table II – Equipment utilization statistics

For all core size designs, the simulated cpu was performing useful work; that is, not idling, for less than 10% of the simulated run time. This apparently low percentage should be viewed in light of the fact that for approximately 66% of the time the cpu is forced to idle because there are no jobs in the system, and therefore the maximum possible cpu utilization in the type of operating environment modeled is approximately 33%. The decreasing cpu utilization with increasing core size is attributed to the reduced requirement for executing program loader and initialization functions.

The disk channel utilization, which represents the percentage of the simulated run time during which commands or data were being transferred via the disk channel, shows a decrease at the increasing core size designs. This reflects the impact of not reading processing programs from the disk at the higher core sizes. Thus, it can be interpreted that for the 24K core size design approximately one-half of the disk channel utilization time is spent in transferring processing programs from disk to core.

The utilization of the magnetic tape channel, which was modeled as a path distinct from the disk channel, was extremely low and constant for all core size designs. The constant 0.2% value is attributable to the fact that of the job types modeled, only the file update job was defined as requiring data transaction recording on the tape. Both the frequency of occurrence of this job type as well as the volume of data transferred was low.

Disk arm utilization, per arm, is relatively constant for all core size designs with the multi-job processing capability. The reading of processing programs from disk arm #2 accounts for the higher utilization at the lower core size designs. Similarly, the disk arm #2 utilization for the 32K core size design with single-job processing is higher than the 32K design with multi-job processing because more program reads were executed to handle the jobs singly than in groups. Disk arm utilization is a valid criterion for evaluating the merits of a disk map relative to the reliability of disk operations. That is for any given core size design, a constant utilization across all four disk arms reflects an evenly distributed work load and minimizes the relative vulnerability of any particular disk arm to failure.

SUMMARY

The primary goal of this effort was to examine from the system user's point of view, the effect of core memory size on system performance. For a dedicated special-purpose type of multiple-access information system such as the one modeled, it was evident that the advantages to be gained through incremental increases in core storage space could only be reflected in the model, as well as in the actual system, by increasing the system's apparent processing capability. The specific interest here was not in validating a specific software design, particularly since at the time the activity was conducted neither a control nor application program solution existed. Therefore only an envisioned design, deemed to be appropriate and responsive to the implied requirements of the system, was modeled on a broad, or macro, level Starting with a 24K word core memory size which was based on certain assumptions of program, buffer, and work space requirements, the capability of the processing system was improved for additional increments of 8K words by increasing the available user work area. In the modeled system, this allowed for more appli-

cation programs to co-exist in core memory and increased the multi-job processing limit.

The most significant characteristic of the system, as demonstrated by the simulation statistics, was the slow job input rate relative to the fast job processing rate; viz., a ratio of approximately 1:3. Since the user terminals were the sole source of jobs into the system, then the man-machine mechanism for entering job statements was most significant in establishing the input rate. This disparity in rates was such that there was no significant queueing in the system at any core size solution. It is evident that the operational configuration represented in the model could withstand substantial change and not appreciably affect the job turn-around time. With the same job types and operational procedures, doubling the number of active user terminals, for example, would only raise the job input rate to approximately two-thirds of the processing rate. With eight user terminals active, halving the number of steps in the job definition sequence or halving the speed of the modeled cpu would have similar effects.

The cpu represented in the model is characteristic of the processing power usually associated with multiple-access time-sharing systems, and on that basis was *a priori* considered to be an appropriate hardware solution. As the run results indicate, however, the cpu overpowers this system's processing problem. A cpu with one-half to one-third of the execution speed modeled (i.e., an average instruction execution time between 20 μ seconds and 30 μ seconds) could still result in an input to processing rate ratio of less than unity, and thus no significant queueing in the system. The processor's response to user input would be slower, of course, but imperceptibly different to the system user.

Based on an understanding of the operational requirements of this system and some prior association with information handling programs, a level of detail was established in the model which would support the objectives of the simulation. Certain quantitative assumptions established the minimum core size to be modeled at 24K words; this was an arbitrary reference point from which to examine the effect of core memory size on system performance. The run statistics showed that as the core size, and hence the processing capability, increased there was no appreciable change in system performance from the system user's point of view. We conclude that not only is the smallest core size modeled adequate, but that lower core memory sizes (i.e., 16K words or less), which would require considerably more program overlaying in the control and user program areas, would provide an adequate and responsive solution. Implicit in this conclusion, is that an improved processing capability, such as multi-job processing in the modeled system, is of little value in a system where the job input rate is considerably slower than the processing rate.

Admittedly, certain of these results could have been determined through analytical methods. This is largely due, however, to the nature of the chosen example system. In general, the system designer's choice of analysis methods should not be delimited by only the immediate considerations, but should instead be made in light of the entire planned system development activity. A simulation program model, even in the case of the cited example system and however gross in its initial form, provides the designer with an important analysis tool which can be modified and refined part by part as the system design phase progresses so as to evaluate and verify the evolving solution.

A multiple-access, information storage and retrieval system in which the job input rate is considerably slower than the job processing rate has been discussed here, whose design requirements contrast rather sharply with those of other multiple-access computing systems. This special purpose system is representative of a type which can be expected to become widespread as data processing extends to the smaller installations and users. The handling of current account data on credit applicants in a merchandise chain or the maintenance of current warehouse inventories by wholesale distributors are examples of such systems which typically would not require extensive processing power. The information storage and retrieval processing requirements of these types of installations as those of our cited military system, can be satisfied through straight-forward programming solutions, implemented on computers with moderate speeds and with modest core memory capacities.

This activity, furthermore, has served to emphasize the value of a macro level modeling approach to a system design problem; which in the authors' opinion is a fact not commonly enough recognized. When employed as early as is reasonable in the design process, it can yield, with an economy of effort, valuable insight to the system under study.

REFERENCES

1 J McCARTHY
Time sharing computer system
In Computers and the World of the Future M. Greenberger ed. The MIT Press Cambridge Mass pp 221-236 1964

2 J B DENNIS
A multiuser computation facility for education and research
Communications of the ACM vol 7 pp 521-529 September 1964

3 A H TAUB
On time-sharing systems, design and use
Proceedings—IBM Scientific Computing Symposium on Man-

Machine Communication pp 9-16 1965
4 R M FANO
The MAC system: the computer utility approach
IEEE Spectrum pp 56-64 January 1965

5 J I SCHWARTZ et al
A general-purpose time-sharing system
Proceedings—Spring Joint Computer Conference pp 397-411
1964

SODAS and a methodology for system design

by DAVID L. PARNAS and JOHN A. DARRINGER
Carnegie Institute of Technology
Pittsburgh, Pennsylvania

INTRODUCTION

SODAS (**S**tructure **O**riented **D**escription **A**nd **S**imulation) is a simulation language and compiler being designed at Carnegie Tech for use as a tool by the designers of computer systems. The structure of the language and its translator reflect a definition of "system" and a methodology for "system design." It is the purpose of this paper to present the proposed methodology and language.

Definition of "system"

We shall propose a definition of "system" to be used within this paper. It is not proposed as a general definition of system, but as a local definition within this paper as an aid in discussing the design and simulation of systems. There may be systems which do not fit the definition, but it will be clear that those items usually called computer systems, either hardware or software, do fit the definition.

A *system* is a connected set of components whose behavior is self determined. By *self determined* we mean that the state and outputs of the system can be predicted from knowledge of the state of the system and the inputs. A *component* is a device or program which is self determined and may, in fact, be a system itself. By *connected* we mean that the inputs to some of the components may be the outputs of other components of the system. It should be clear that the distinction between system and component is one of level rather than substance. Any system may be a component of some other system. Ultimately, the components of the system are basic units of some sort. The only requirement that we place on these basic units is that they have well defined behavior, i.e., that it must be possible to find an algorithm which describes their behavior.

A system design methodology

A principal point in most courses in engineering analysis is the importance of having a precise definition of a problem before attacking it. Throughout such courses or textbooks this appears as a recurrent theme: A precise definition of the problem is a prerequisite to the complete understanding of the problem by the designer. A precise definition of the problem is the first step in finding the proper structuring of the problem into sub-problems. A precise definition of the problem is necessary to the evaluation of proposed solutions.

In the design of complex systems, such as computer systems, precise definitions of the goal of the design process are not easy to find. As a particularly simple example, consider the problem of designing a special purpose traffic control computer for the single intersection shown in Figure 2. Certain aspects of the problem are easy to define. The nature of the inputs to the computer can be specified, e.g., "20 inputs from sensors: 1 volt indicates the presence of a car, 0 volts the absence of a car." The nature of the outputs (e.g., the lights to be controlled) can also be specified. It is very difficult to specify the behavior desired of the computer. One can discuss the design criteria for the behavior of the computer, e.g., it should minimize average wait time or maximize the traffic handling capacity. One can place restraints on the behavior, e.g., the maximum time for lights to remain in any mode is three minutes. Such statements do not specify the behavior; they constrain the behavior of the system. One of the design principles motivating the design of the SODAS system is that the behavior should be defined before the design of the computer itself is started.

The behavior of the system can best be described by an algorithm.* Certain variables in the algorithm can be designated as inputs or outputs, the behavior of the system described being the output response to each permissible string of input values. A designer who begins the design of a computer or similar system without such a specification has set out to solve an un-

*The algorithm need not be a computer program.

specified problem and, in all likelihood, he will design a system with suboptimal behavior (because he will confuse the question of behavior with the question of how to construct the system). The availability of an algorithm specifying the behavior makes it possible (1) to experiment (by simulation) with behavior explicitly (thereby determining exactly what is optimal behavior), and (2) to compare a proposed design with the specified behavior (to make sure that the system, as designed, accomplishes its goals).

After the behavior of the system is specified, a further structuring of the problem is desired. This usually means a specification of the various sub-systems or components and the way in which they are connected. Each of the sub-components will then become a design problem in itself. Two problems arise: the first is specifying the behavior expected of the sub-component; the second is determining that the specified sub-components, working together, will actually result in correct behavior for the whole system. Every system design team has had the experience of bringing together separately designed components only to find either (1) that the specifications for some component had been ambiguous or misunderstood or (2) that the components actually did not work together, although they did meet the specifications. The behavior of the sub-components may also be specified by algorithms. If desired, these algorithms could be verified by simulating the set of connected components and comparing the behavior of the whole system to the original specification of its behavior.

Each of the sub-components can then be broken down in the same way, until units small enough to be completely designed are obtained.

We call this approach to system design the "top down" approach. It involves starting at the "top" with a complete specification of the desired behavior of the system, then breaking the system down into smaller and smaller components until the system is specified in terms of the basic building units.

If the specifications are in languages for which computer translators are available, then it is possible to make use of simulation throughout the design process (to verify that the specifications correctly indicate the desired behavior). The specifications at every level provide evaluation criteria for the next lower level.

It is probably worthwhile noting the most obvious and important advantage of using simulation throughout all levels of the design problem. Often when the design has reached a level of great detail, decisions made at higher levels demonstrate themselves to be inconvenient and costly. If the lower levels of detail are reached only when the design has progressed to the point where a hardware implementation has begun, it is often too late to go back and make changes. If the simulation language allows the design to proceed to great detail, *entirely in simulation,* a new feedback loop is added to the design process. The detail design of one component can be allowed to influence a higher level design of that component or other components. The result should be a better design with fewer last minute changes.

Properties of the specification-design language and translator

In the following paragraphs we shall attempt to list and discuss the features which must be present in a specifications language and simulation system if it is to be a useful tool for "top down" system design. In a later section of the paper we shall discuss our current attempt at producing such a language and show that it provides the necessary features.

(1) Designation of inputs and outputs

It is necessary to specify which variables in the algorithm describing a system are inputs and outputs. If these are not distinguished, the system is likely to be overspecified (since the designer will have to produce a component that will duplicate the behavior of the algorithm on all its variables, not simply those which will be used as inputs and outputs).

(2) Combination of independently written descriptions

The language must allow the description of systems of components which have been separately described. It must be possible to take separately written algorithms, indicate the way that inputs of one are connected with the outputs of others, without excessive worry over conflicts in names of variables, etc.

(3) Correct handling of simultaneous events

The translator for the language must be capable of correctly simulating simultaneous events in a system. If two of the separately described components (as mentioned above) happen to be active at the same time and interact closely, the translator must correctly simulate these simultaneous events (although it is restricted to serial execution of the individual algorithms). A method for doing this has been described in detail elsewhere.[1]

(4) Components which are themselves descriptions of systems

The language structure must be recursive; i.e., any system described in the language must be acceptable as a sub-system of a system to be described in the language. To phrase it another way, all the features available to the designer for describing the whole system must be available for describing the sub-systems.

(5) Descriptions with mixed levels of detail

The language and translator must permit the description of components of the system in varying levels of detail. The design of one component may advance faster than the design of the rest of the system. It should be possible to combine a detailed description of one component with less detailed specifications of others for testing purposes. Without this the detailed testing of components in large systems would be impossible.

(6) Mixed structural and behavioral descriptions

The language must allow structural or behavioral descriptions, where appropriate. A structural description of a system describes it as a set of components and their interconnection; a *behavioral* description is an algorithm which duplicates the behavior of the system. At various stages in the design process there is need for structural descriptions, behavioral descriptions and mixtures of both. The language must be designed to permit this. (The ultimate goal of the design process is usually structural description down to a low level of detail, but the description of the components at the lowest level is behavioral.)

(7) Broad class of systems

The language must allow the description of both synchronous and asynchronous discrete systems as well as analog or continuous systems and hybrid systems.

(8) Variety of languages for component description

The descriptive language and simulation system should allow a variety of means of describing the various components in the system. There are already many general purpose and special purpose simulation languages. Each of these has its own strengths and weaknesses. The language most appropriate for the description of one component may not be the same as the language most appropriate for the description of other components. It is a desirable (though not strictly necessary) feature of a simulation system that it permit the description and simulation of systems whose components are described in quite different languages. Often considerable space may be saved by simulating a part of the system in space-taking detail and the remainder at a higher level which requires less space.

Other significant design goals for the language which are not peculiar to the problem discussed here include:

(1) Small local changes in the system being described should require only small or local changes in the descriptive program.

(2) It should not be necessary to provide duplicate descriptions of duplicate components or even of components that differ only in the values of parameters.

A description of SODAS

This section is a description of the SODAS language. Several preliminary comments are in order:

(1) SODAS is an experimental language and is in the midst of its first experimental implementation. This results in several problems:
   (a) Some of the sub-languages to be used in the system are not as yet defined. The grammar contains several terminal symbols indicating descriptions in other languages. These will eventually become non-terminals in the language.
   (b) The language has not received actual use. Use will undoubtedly reveal restrictions and omissions in the language which will have to be removed.

   We have chosen to present the language at this time in spite of its unfinished implementation because we felt that the philosophy underlying the system is sufficiently different from the underlying philosophy of other simulation languages to be worth communicating in itself. Further, we hope that interested readers not involved with the project will give us the advantage of their different viewpoints with constructive criticism.

(2) Our preliminary implementation of the language uses a slightly extended version of the Wirth-Weber precedence analyzer and parser which was described elsewhere[2,3]. The class of grammars for which this analyzer is useful is only a proper subset of the class of context free grammars. In order to use the analyzer the grammar has been distorted from what might seem a more natural grammar for the language. Readers who wonder about certain parts of the grammar where the construction seems a bit strange can attribute the strangeness to our attempts to force the grammar into a restrictive mold. While it would have been possible for us to use a different grammar for expository purposes than we are using in driving our implementation, such an approach seems to introduce too many possibilities for error and has thus been avoided.

The main points contained in the grammar shown in Figure 1 are:

(1) A SODAS system is a set of sub-systems, each with specified inputs and outputs, together with a "wiring diagram" description of the way that

the components communicate.

(2) There may be only one sub-system—the system itself—or any number of sub-systems.

(3) The sub-systems may be described in any language which has been implemented in the system, including the SODAS language itself. This is a property of the simulation algorithm that is the basis of SODAS. The algorithm depends only on the existence of algorithms for simulating the sub-systems and not at all on the language in which the algorithms were originally described.

```
<SODAS descr> ::= <SFD descr> | <BOOLE descr> | <other descr> |
     SODAS begin <dec set> <c-dec set> <components> end |
     SODAS begin <dec set> <components> end
<dec set> ::= <decs>
<decs> ::= <wdec> | <decs> ; <wdec> | <decs> ; <i/o dec> | <i/o dec>
<type> ::= integer | real | Boolean | register
<idlist> ::= identifier | <idlist> , identifier
<idset> ::= <idlist>
<idec> ::= input <type> <idset> | input <type> array <idset> <bounds>
<odec> ::= output <type> <idset> | output <type> array <idset> <bounds>
<wdec> ::= <type> <idset> | <type> array <idset> <bounds>
<reference> ::= identifier | <reference> [ <paramlist> ]
<refer> ::= <reference>
<dep dec> ::= <refer> ← . <refer>
<dep decs> ::= <dep decset>
<dep decset> ::= <dep dec> | <dep decset> ; <dep dec>
<b-pairset> ::=<constant>:<constant>| <b-pairset> ,<constant>:<constant>
<b-pairs> ::= <b-pairset>
<bounds> ::= [ <b-pairs> ]
<comp dec> ::= subsystem <c-ident> ( <idset> ) <i/o dec set> <dep decs>
     <comp body> |
     subsystem <c-ident> <i/o dec set> <dep decs> <comp body> |
     subsystem <c-ident> <i/o dec set> <comp body> |
     subsystem <c-ident> ( <idset> ) <i/o dec set> <comp body>
<comp body> ::= <SODAS descr> | clock<constant><SODAS descr>
<component> ::= subsystem <i/o dec set> <connect set> <comp body> |
     subsystem <i/o dec set> <connect set> <dep decs> <comp body> |
     subsystem <c-ident> ( <paramlist> ) <connect set> |
     subsystem <c-ident> <connect set>
<c-ident> ::=<identifier>
<comp set> ::= <component> | <comp set> ; <component>
<components> ::= <comp set>
<con grp> ::= <refer> ←<refer> | <refer> →<refer> |
     <con grp> ∧ <refer> →<refer> | <con grp> ∧ <refer> ←<refer>
<connect set> ::= <con grp>
<params> ::=<identifier>|<constant>| <params> ,<constant>|
     <params> ,<identifier>
<paramlist> ::= <params>
<c-dec set> ::= <c-decs>
<c-decs> ::= <comp dec> | <c-decs> ; <comp dec>
<i/o dec> ::= <idec> | <odec>
<i/o dec set> ::= <i/o decs>
<i/o decs> ::= <i/o dec> | <i/o decs> ; <i/o dec>
```

Figure 1—Grammar for SODAS

There is, however, much information about the language which cannot be contained in the syntax.

The language is designed for use in simulating discrete systems. These are systems which can be considered as changing state only at discrete points in time known as "clock pulses." As is always the case in simulation on digital computers, systems whose state variables are changing continuously over periods of time can be simulated only to the extent that they can be approximated as discrete systems.*

The algorithms describing the behavior of a system or component describe the behavior on each clock pulse. The variables in the algorithm are the memory of the component, the information that it carries from clock pulse to clock pulse. Inputs are read and outputs computed on each clock pulse. It is, however, possible for an algorithm to indicate that it will be inactive for a fixed number of clock pulses or until certain external conditions hold. The simulation system can take account of this in determining an efficient way to simulate the system.

To correctly simulate simultaneous events by simulating the individual events in sequence, it is sometimes necessary to simulate one component, restore it to its previous state, and later simulate it again.

Therefore, the simulation algorithm requires of a sub-language that its compiled algorithm can be simulated without a permanent change in the state variables or memory of the algorithm. Since this is the only firm requirement of sub-languages, it is possible for sub-language compilers to be written independently of the SODAS compiler itself.

We shall now consider the interpretation of the various syntactic units of the language.

SODAS DESCR: A SODAS description may be a description in any of the various other languages included in the system, or it may have the form:

SODAS BEGIN <DEC-SET> <C-DEC SET> <COMPONENTS> *END*

The terms in italic are terminal symbols in the language and have no syntactic definition in the grammar.

DEC SET: A DEC SET is a set of declarations much like ALGOL declarations. Variables may be declared as inputs, outputs, or simply interconnecting variables. Those declared to be inputs and outputs are for use in communicating witth external systems; the others are used later in describing the way that the components are connected. They may be thought of as patch cords which are used to connect the output of one component to the input of another.

C-DEC SET: A C-DEC SET is the set of component declarations. Any component or type of component which will be used more than once in the system can be declared here and given a name and optional parameters. The name can be used later as a substitute for a description of the component. If the parameters are given, then the declaration is a description of a class of components. A member of this class is determined by supplying values for the parameters during the use of the component. Separate uses of the component or members of the class of components are completely independent. They have no common memory or interconnectors. They are simply copies of a common template.

The form of a component declaration is:

SUBSYSTEM <C-IDENT> (<IDSET>) <I/O DEC SET> <DEP DECS> <COMP BODY>

C-IDENT: The C-IDENT is the identifier assigned to the component or class of components. The IDSET is simply the (possibly empty) list of parameters for a declaration which defines a class of components. The I/O DEC SET is a set of declarations (as described above) which indicate the inputs and outputs of the component.

DEP DECS: The set of statements known as DEP DECS is a description of some basic properties of the component. It is possible in SODAS to deal with devices that have instantaneous response. An output of a device at some discrete point in time can be a function of the inputs to that device at the same time. SODAS is capable of simulating such components and systems correctly, but it must have information about such immediate dependencies.

A DEP DEC is a statement of the form:

(Output) (Input)
IDENTIFIER ←• IDENTIFIER

The variable on the left must be an output of the component, the variable on the right an input to the component. Such a declaration indicates that the value of the component on the left *may* be a delayless function of the input indicated. If the statement is not true it may result in a less efficient simulation than necessary, or in some cases in the SODAS system's deciding that the system cannot be simulated. An unnecessary statement will never produce an incorrect simulation, but a missing statement might do so. It is planned to allow such declarations to be conditional in future implementations—this will expand the class of systems that SODAS can deal with. At present the syntax only allows unconditional declarations.

The component body is a system description, either in SODAS or in some other system

*This restriction is fundamental to the problem of simulating continuous change on a digital computer. It is not a special restriction in SODAS, nor is it a disabling restriction. We are as close to achieving design goal (7) as is possible.

COMPONENTS: The syntactic type COMPONENTS is the set of all the parts of the sub-system. Each component is either an instance of a declared component type or is described in the body of the system description. Components which are instances of a type already declared are specified by the identifier used in their declaration and the values of the parameters, if any. The declaration and body of other components appear in the system description. As each component is described, its connections are also described. A connection is described by statements of the form

(Input) (Interconnector)
IDENTIFIER ← IDENTIFIER

where the identifier on the left is an input variable for the component, and the identifier on the right is an interconnector (or wire) in the system, or by a statement of the form

(Output) (Interconnector)
IDENTIFIER → IDENTIFIER

where the identifier on the left is an output of the component and the one on the right is an interconnector.

By consistently placing the interconnector on the right side of the statement, it is possible to resolve any possible conflicts between the names of interconnectors in the system and inputs to the component.

In the following we find it convenient to use the term "parent" system to describe the smallest system which contains a given system as a component. Any system which contains the given system as a component may be termed an ancestor.

Timing

It was mentioned earlier that each component of a system is described by an algorithm which performs the actions taken on a single clock pulse each time it is executed. The most straightforward method of simulating such systems would be to execute every algorithm on every simulated clock pulse. Such a simulation would often be a very inefficient simulation of a system. SODAS has several descriptive features which avoid such simulations where possible.

The systems simulated in SODAS are discrete systems in that activity is presumed to occur only at discrete points in time. The system as a whole can be thought of as being driven by an external clock pulse, but the clock pulses need not be thought of as equally spaced in time. There must be a clock pulse whenever activity occurs, but these periods of activity need not be at integer multiples of some basic time unit. This facilitates simulation of systems whose sub-systems are not naturally described on the same time scale.

Systems in SODAS may be either synchronous or non-synchronous. In a synchronous system all subsystems or components are assumed to be clocked by a master clock of the system. Activity takes place only on clock pulses which occur once every basic time unit. For each sub-system, the time between clock pulses is either the same as that of the parent system or some integer multiple of it.

In non-synchronous systems each component determines its own periods of activity and inactivity. If all components of a system are inactive, the system is inactive.

Both types of system may be simulated in SODAS. A SODAS system is synchronous if all of its sub-systems are synchronous. A non-synchronous system may have synchronous sub-systems, but a synchronous system may not have non-synchronous sub-systems.

Both synchronous and non-synchronous components may be described as having a clock interval which is a constant multiple of the clock interval of the parent system. For synchronous systems the clock interval of a component must be an integer multiple of the clock interval of the system. For non-synchronous systems the interpretation of the clock rate description is quite different. Each non-synchronous component must keep track of the simulated time. The clock rate description simply indicates a difference in scale between the time measures used within the component and that used in the parent system.

Both synchronous and non-synchronous systems may have components which indicate that they are to be inactive for periods of time. For a synchronous system the inactive periods must be integer multiples of the basic time unit, but this restriction is not needed for non-synchronous systems.

Both synchronous and non-synchronous sub-systems may indicate that they will be inactive until some Boolean expression involving inputs holds. This is equivalent to the **wait until** of SOL.[5]

A system is active only if at least one of its components is active. Thus, if all components of a system are inactive, the system need not be simulated. If one or more components are doing a "wait until," and their Boolean expressions include inputs external to the parent, then if all sub-systems are inactive, the parent system must report to the ancestor system that *it* is doing a "wait until," and indicate the variables involved.

It is hoped that the above provides a sufficient set of mechanisms to allow efficient simulation of subsystems which are essentially inactive (either marking time or waiting for an external event). With this feature, one can approach the efficiency of systems like GPSS[10] and SIMSCRIPT.[11] The unique feature of the simulation method used in SODAS is its behavior when several components are active at once. At this point SODAS can determine a correct order for simulating the sub-systems.

We are including an example to demonstrate the use of SODAS. Since the SODAS system is not yet operational, we have had to simulate its behavior using Carnegie Tech's ALGOL and BOOLE[6] translators. The problem is not claimed to be a practical problem, nor is it touted as an example of a particularly good design. It was chosen because it was small enough for presentation and assimilation in a relatively short period of time, yet large enough to illustrate the features of SODAS. The example is presented in considerable detail so that those who wish to study the way that the language is used may do so.

The problem that will be discussed was first presented in a Carnegie Institute of Technology course on logic design in 1963. In this course, the students must completely design a working system in terms of idealized logic. The problem constitutes a term project for groups of three students who were seniors in Electrical Engineering. In 1963, the majority of the groups completed the design in the relatively short period of time alloted. SODAS was not available to the students.

This project, and those of later years, gave the senior author the opportunity of observing a large number of design teams working on the same design project. This observation led to a number of conclusions:

(1) Groups which used the "bottom up" approach, designing small parts of the project and trying to put them together into a total system, did not produce working systems.

(2) Among the groups that were successful, the work was generally quite poor, unstructured, and incorrect, until they either hit upon (or were led to) the top-down approach. All work before that could be classified as false starts.

(3) After the top-down approach was started, the work progressed well until it became time to test the unit composed of the components designed separately by the various members of the group. There were two distinct sorts of difficulties:

(a) Components which should have worked together did not. The students did not have a precise way of communicating the specifications of the components to each other. When they came down to the moment of truth, it was apparent that each had his own idea of the division of labor. As a result, extensive last minute redesign was required. A few groups did not complete the project due to this factor.

(b) Even though the individual components met their specifications, the combined system failed. The difficulty was that the students had no means of verifying that their initial structuring of the problem was correct and feasible. Some oversight at the earlier "behavior specification of sub-components" stage had not been detectable until the components were designed and being tested. These errors were often very difficult to recover from, and the redesigned systems that did work often were obtained at the price of poor performance by the system.

We have no doubt that the SODAS system would have been a great deal of aid to these students—enough aid to turn what was, for them, a very difficult problem into a relatively easy problem. The authors feel quite strongly that such problems are not confined to students in courses, but are common in professional circles as well.

The problem described below is deceptively simple. Assigned to the class mentioned, without SODAS, it was a problem that was probably too large for the time allowed to the students. With the aid of SODAS and the methodology discussed above, it now appears far too simple for the course. This may only be a matter of appearance, resulting from our increased familiarity with the problem; only actual experiments in design with the aid of SODAS will test our conjecture that the increased simplicity of the problem results from the availability of the new system.

An example: design of a traffic control module

A modular design of a traffic control system would include traffic control modules at each intersection and a master control unit to supervise the overall operation. In the following example we are going to look at possible specifications for an individual module and follow through the design of some of the actual hardware—using the "SODAS Method" of system design.

Specifications for traffic control module

Each module is to control a single intersection such as is shown in Figure 2 and be easily adaptable to a simpler intersection. It is to operate with various degrees of influence from the master unit, ranging from complete outside control to independent operation. In addition to controlling the traffic lights, the module is to supply information about the local traffic conditions to the master unit.

To limit the complexity of this example, the module will use only the two sets of modes (traffic flow patterns) shown in Figure 3. The first set (modes 1 through 4) is to be used if the intersection does not have a separate left turn lane. Otherwise, the second set (modes 5 through 8) is to be used.

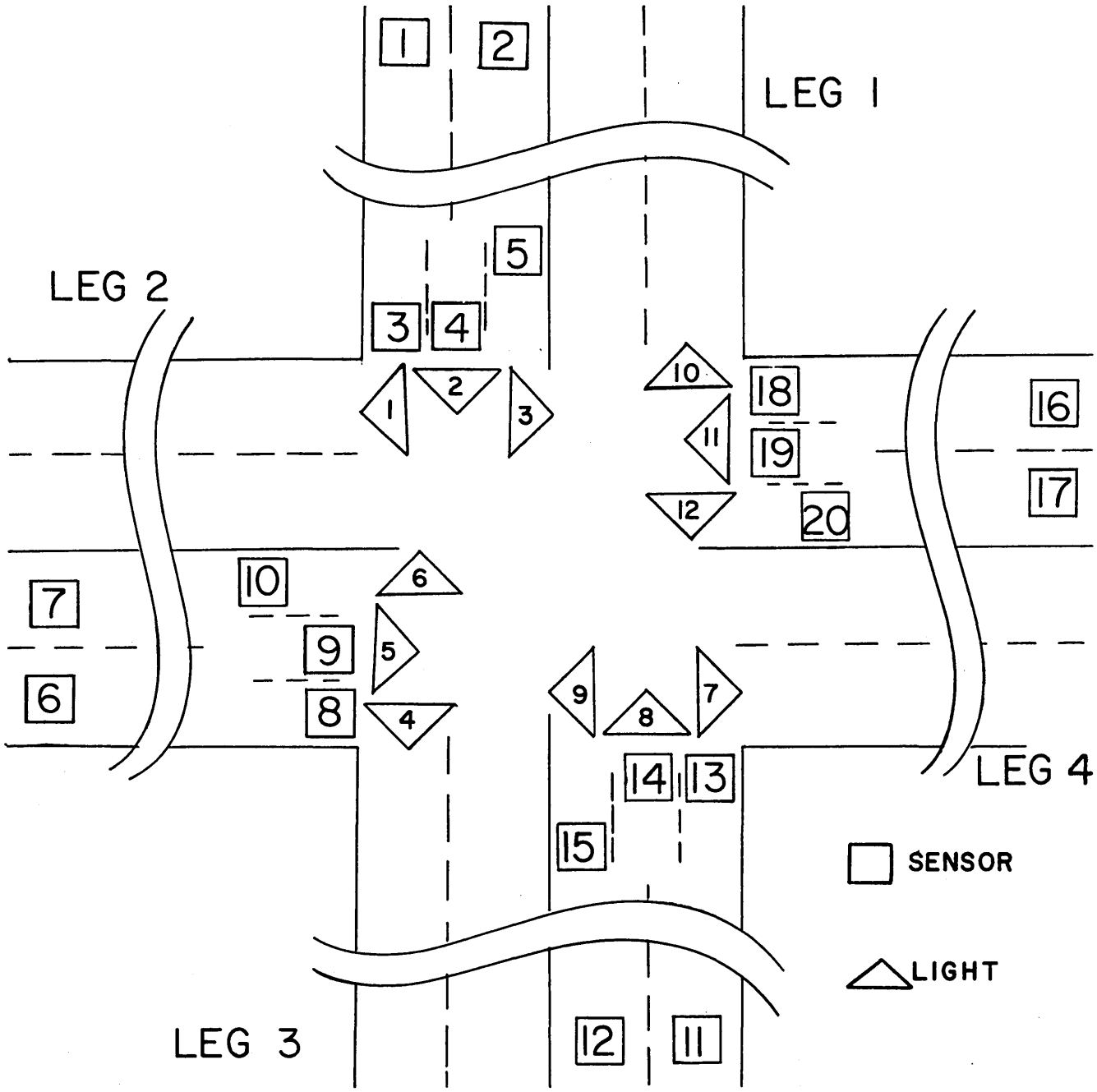


Figure 2—Model intersection

Control module inputs (from sensors and master control unit)

SENSOR [1:20] —Sensor inputs (See Figure 2.)

OUTSIDELIGHT [1·12] —Master unit's control lines for the lights

TMAX —Maximum time in any mode (traffic flow pattern)

TMIN —Minimum time in any mode

CYCLETIME —Total time to cycle through all 4 modes

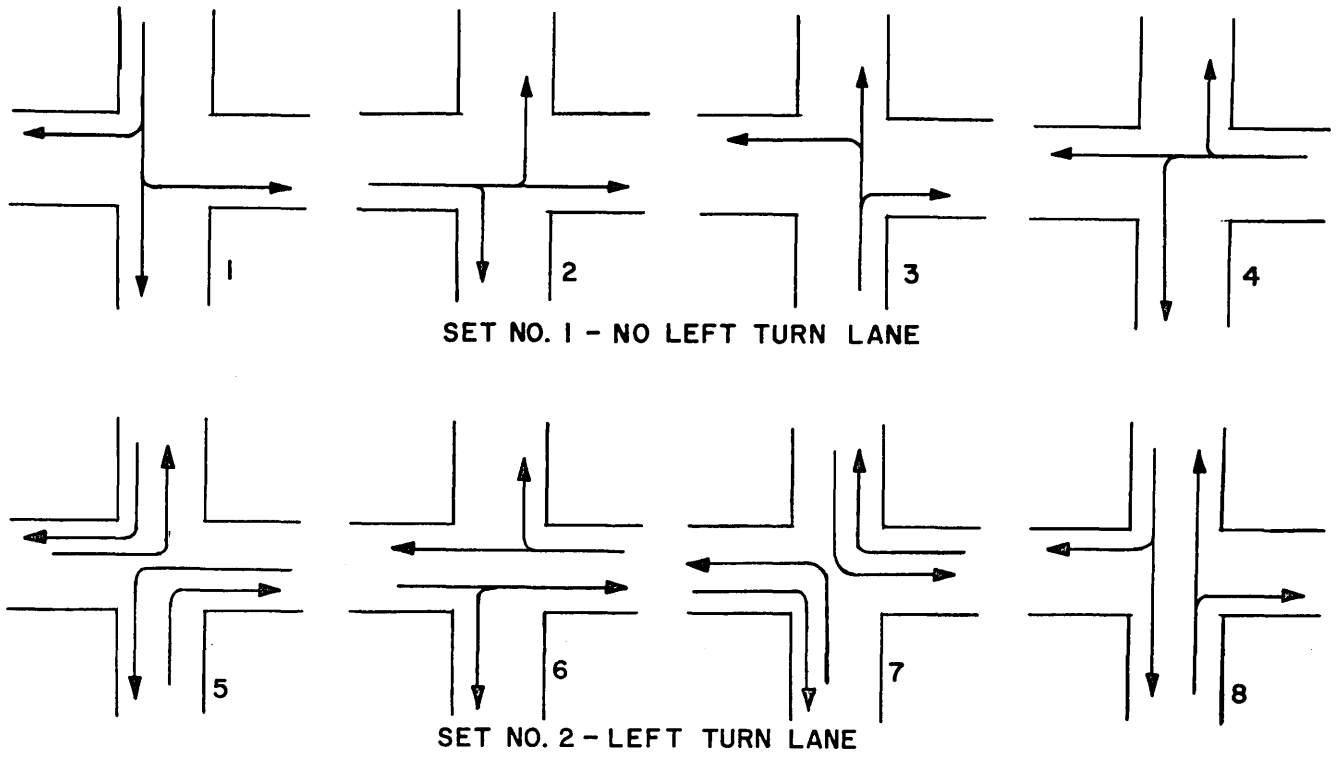


Figure 3—Traffic flow patterns

INITIALMODE	—Initial mode
COC	—Complete outside control switch
LOC	—Limited outside control switch
RESET	—Initialization switch

Control module outputs (to lights and master control unit)

CARS [1:4]	—Number of cars waiting in each leg of intersection
LEFTTURN [1:4]	—Lines indicating whether cars are waiting in left-turn lanes
LIGHT [1:12]	—Signals controlling the 12 traffic lights.

Through inputs COC and LOC the master unit indicates one of three possible degrees of outside control: complete outside control, limited outside control, independent operation. When the module is operating independently, it is expected to keep account of the number of cars waiting to pass through the intersection and to control the lights appropriately. The control is, however, to be subject to two parameters, the minimum time and maximum time to spend in any one mode (condition of the traffic lights). This will assure the driver of a minimum amount of time to pass through the intersection, once he receives a "green light," and also assure him of a maximum waiting time (when he is on a little-used road intersecting one which is very heavily used).

The LOC input signal indicates that the module is to leave its completely independent operation and submit to limited outside control. In this type of operation, the module is still partially in control, but is subject to a basic cycle time imposed by the master unit in order to synchronize it with other units. The basic cycle time is the total time in which the unit must complete its cycle of all four modes shown in Figure 3. At this time, of course, the master unit can also set the maximum and minimum times to values which will give the module the degree of freedom desired. At the time of initiating outside control the master control unit indicates the initial mode for the module, thus bringing about complete synchronization. Provision is also made for direct control of the lights by the master unit (OUTSIDELIGHT [1:12]). For each of the 12 lights at the intersection, a line is provided to signal TRUE for green and FALSE for red. It is assumed that the traf-

fic lights have mechanical timers that control the yellow light during the changes.

The clock frequently is to be 10 Hz. Although unusually slow, it assures that all cars will be detected correctly at speeds less than 100 mph.

First level of design

The first step taken was to treat the module as the black box shown in Figure 4 and to write a SODAS program describing its inputs, outputs, and behavior. This initial description was written in SFD-ALGOL,[7] an ALGOL-like sub-language of SODAS with facilities for specifying inputs, outputs, and timing.

In conjunction with the above, a traffic simulator was written. Such a simulator is most easily described in a SODAS sub-language resembling SIMULA[12] and is treated as a component in the total system.

The entire system, including the traffic, was then simulated and changes were made in the algorithm that selects the traffic flow pattern until reasonable traffic flow was obtained. Since, at this stage, the control algorithm was described in an ALGOL-like language and not embedded in hardware, such testing and changing was a simple task. Figure 5 contains the SODAS description of the traffic control system at this point.

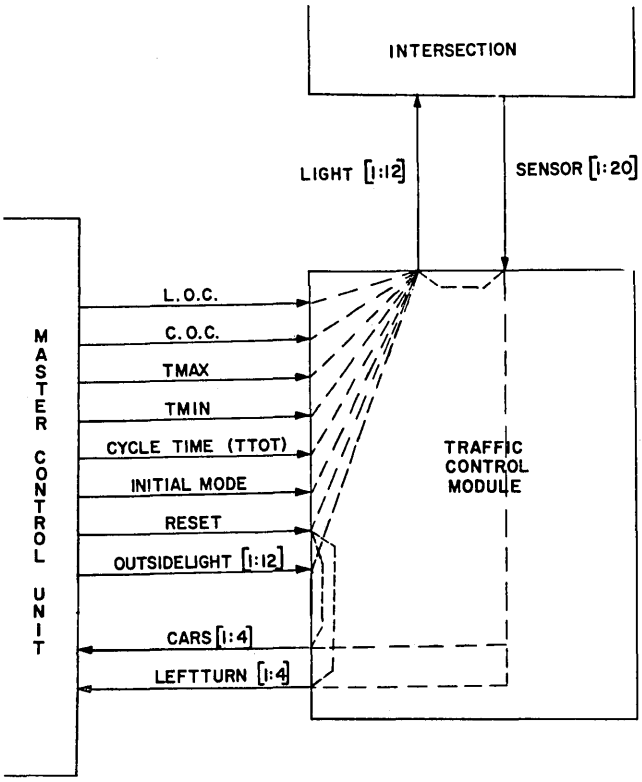


Figure 4—First level of description

Second level of design

At the second level of design, structural information was added to the behavioral description obtained above.

The module is required to count the cars waiting in each leg of the intersection at all times. Since this task is identical for each leg and since such a counter is easily isolated from the rest of the system, it was decided to design four counters as sub-systems.

The single sensor in the left-turn lane makes the counting of cars more involved. It is assumed that if a car were waiting in the left-turn lane and there were a green light on the last clock pulse, it would leave on this clock pulse. This assumption forces the counter for each leg to record the "left-turn light" to account for cars leaving the leg through the left turn lane.

The module initially selects its traffic flow pattern from the first set (Figure 3). It assumes that no left-turn lane exists until one of the left-turn sensors is activated. These signals are combined by a "joiner" (actually an "or" gate). Figure 6 indicates the structure introduced at this point.

In the first level of design, the task of counting cars was handled by the procedure "count." However, as written, this procedure had to be executed twice for each simulated clock pulse—once to update the car-count and again to store the value of the left-turn light after it was determined.

At the second level of design, this problem is reflected in the fact that the values for LIGHT [1:12] (counter input) are declared immediately dependent on CARS [1:4] (counter output). This immediate dependency and others are indicated in the block diagram (Figure 6) by dotted lines and in the SODAS description (Figure 7) by dependency statements.

The reader should note the close correspondence between the block diagram and the SODAS description. Since the four counters are identical, only one needs to be described (by a component declaration). Then the four counter descriptions are references to that declaration.

Third level of design

The third step in the design is the final step in our example. In it the counter for leg 1 is divided into four sub-systems and each of these is implemented in idealized logic elements (flip-flops and gates). Although we could have introduced one more level by describing the components of the counter in SFD-ALGOL, these descriptions would have been extremely simple and it seemed more reasonable to implement them directly using logic elements.

Of the five sensor inputs to the counter for leg 1, two (1, 2) indicate cars entering the leg and two (3, 4)

```
SODAS Description                                          Comments

SFD begin
input Boolean array sensor [1:20];
input Boolean array outsidelight [1:12];
input Boolean loc, coc, reset;                             declaration of
input integer tmax, tmin, ttot, initialmode;
output Boolean array light [1:12];                         input, output and
output Boolean array leftturn [1:4];
output integer array cars [1:4];                           internal variable
integer mode, tim, ctmax, ctmin, cttot;
Boolean ltl;
real factor;

procedure initializecontroller;
   begin
   integer i;
   for i←1,2,3,4 do                                        initializes
      begin
      cars[i]←0;                                           internal
      leftturn [i]←false;                                  variables
      end;
   for i←1 step 1 until  12 do light[i]←false;             (memory) and
   mode ← initialmode;                                     output
   tim←tmax;  factor←1;
   ctmax←tmax;  ctmin←tmin;                                variables
   cttot← if loc then  ttot else 1800;
   ltl←false;
   end ititialization of the controller;

procedure count (a); Boolean a;
   begin
   own Boolean array lastleftturn [1:4];                   if the parameter
   integer i,j;                                            a is true, the
   for i←1,2,3,4 do                                        number of cars
      if a then                                            waiting on each
         begin                                             leg is updated
         for j←4,3 do if sensor [5*i-j] then               and a left-turn
            cars [i] ← cars [i] + 1;                       lane is checked
         for j← 2,1 do if sensor [5*i-j] then              for.  if a is
            cars [i] ← cars [i] -1 ;                             false, the
         if sensor [5*i] then leftturn [i] ← ltl ← true;         values of
         if leftturn [i] ∧ lastleftlight [i] then                the left-
            begin                                          turn lights are
            cars [i] ← cars [i]-1;                         recorded to be
            leftturn [i] ← false;                          used on the next
            end;                                           clock pulse.
         end else lastleftlight [i] ← light [3*i];
   end count ;
```

Figure 5—SODAS description of traffic control module at level one

(Pts I, II, III)

SODAS Description

```
procedure setlights (a,b); Boolean array a;
 integer b;
   begin
   integer i;
   for i←1 step 1 until 12 do a[i]← false;
   if b=5 then a[ 1]←a[ 6]←a[ 7]←a[12]←true else
   if b=6 then a[ 4]←a[ 5]←a[10]←a[11]←true else
   if b=7 then a[ 4]←a[ 3]←a[10]←a[ 9]←true else
   if b=8 then a[ 1]←a[ 2]←a[ 7]←a[ 8]←true else
   if b=2 then a[ 1]←a[ 4]←a[ 5]←a[ 6]←true else
   if b=4 then a[ 7]←a[10]←a[11]←a[12]←true else
   if b=1 then a[ 1]←a[10]←a[ 2]←a[ 3]←true else
   if b=3 then a[ 4]←a[ 7]←a[ 8]←a[ 9]←true;
   end setlights;
```

Comments: sets the traffic lights given a desired traffic flow pattern (mode)

```
integer procedure newmode;
   begin
   integer i, high, nscars, ewcars, tcars;
   nscars←cars[1] + cars[3];
   ewcars←cars[2] + cars[4];
   tcars←ewcars + nscars;
   if tcars=0 then tcars←1;
   if ¬ltl then
      begin
      high←0;
      for i←1,2,3,4 do
         if i ¬ = mode ∧ (cars[i]>0 ∨ loc) then
            begin
            if cars[i]>high then
               begin
               high← cars[i]; newmode←i;
               end;
            end;
         factor←high/tcars;
      end else
      begin
      if mode ¬ =5∧mode ¬ =7∧(leftturn[1]∨leftturn[3]) then
         begin newmode←7;factor← 5*nscars/tcars;end else
      if mode ¬ =5∧mode ¬ =7∧(leftturn[2]∨leftturn[4]) then
         begin newmode←5;factor← 5*ewcars/tcars;end else
      if mode ¬ =8 ∧ (nscars>ewcars) then
         begin newmode←8;factor←nscars/tcars;end else
         begin newmode←6;factor←ewcars/tcars;end;
      end;
      end of new mode procedure;
```

Comments: selects new mode to maximize the number of cars allowed to move

```
integer procedure cycletime;
   if mode = 5 ∨ mode = 7 then cycletime ← tmin
   else if ¬ loc then cycletime ← tmax
   else cycletime ← factor*cttot;
```

Comments: determines the time to be spent in the new mode

```
SODAS Description                                          Comments

continue: time begin                                       — start of clock pulse
   if reset then initializecontroller;
   if coc then
      begin integer i;                                     under complete
      for i←1 step 1 until 12 do                           outside control
         light [i] ← outsidelight [i];                     lights as set
      go to done;                                          by master
      end;                                                 control's input

   if loc then cttot ← ttot else cttot ← 1800;             under limited
                                                           outside control
                                                           the master unit
                                                           sets the total
                                                           cycletime

   tim ← tim + 1;                                          tim is time in mode

   count (true);                                           update car count

   if tim > ctmax then
      begin                                                after specified
      tim ← 1;                                             time select new
      mode ← newmode;                                      mode, new cycle-
      ctmax ← cycletime;                                   time, and store
      setlights (light, mode);                             new light values
      count (false);                                       for counter
      end;

   done: go to continue;
   end time block;                                         — end of clock pulse
end SFD ALGOL description of traffic
   control module
```

indicate cars leaving. SENSOR [5] also indicates cars leaving but only if the left-turn light is green. Thus at a single clock pulse it is possible for the net change in the number of cars waiting to be +2, +1, 0, −1, −2, −3.

In looking for a structure for the counter, the following was noted:

1. A binary to decimal conversion would be needed to allow communication between the binary outputs of the counter and the decimal inputs of the rest of the system.
2. A special counter would be needed that could accept increments of 1 or 2 and decrements of 1, 2, or 3 in a single clock pulse.
3. A network would be needed to decide
   (i) if a left-turn lane existed
   (ii) if cars were waiting in the left-turn lane
   (iii) if a car had left via the left-turn lane
4. A decoding network would be needed to interpret the signals from the sensors and compute the net change in number of cars waiting.

Figure 8 shows the block diagram of the counter for leg 1 incorporating the structure discussed above.

The "special counter" is a 7-bit counter that ignores inputs attempting to take its value below zero or over 127. This was done to limit error due to incorrect inputs or inadequate counter capacity.

The binary to decimal conversion was described in SFD-ALGOL while the special counter, decoder, memory unit, and the joiner (Figure 6) were designed using standard logic design techniques and described in BOOLE,[6] another sub-language of SODAS.

Figures 9 through 12 show the structural diagrams of the three counter components implemented in logic elements. Studying these figures and the SODAS description of the memory unit at this level in Figure 13,

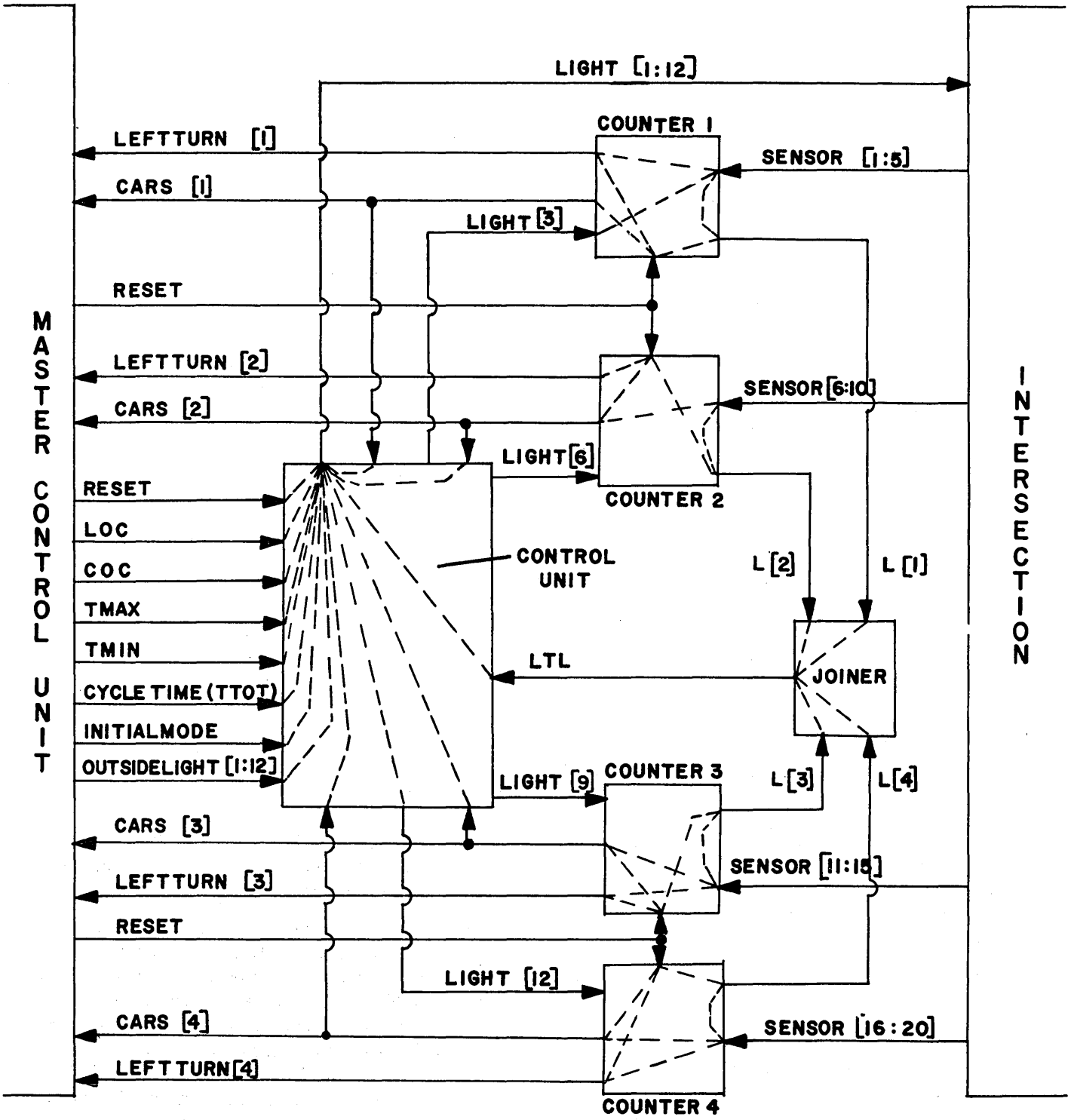


Figure 6—Second level of description

SODAS Description | Comments

```
SODAS begin
   input Boolean array sensor [1:20];
   input Boolean array outsidelight [1:12];
   input Boolean loc, coc, reset;
   input integer tmax, tmin, cycletime,
      initialmode;
   output Boolean array light [1:12];
   output Boolean array leftturn [1:4];
   output integer array cars [1:4];
   Boolean array L[1:4]

subsystem counter

   input Boolean array sensor [1:5];
   input Boolean reset, light;
   output Boolean ltl, leftturn;
   output integer cars
   ltl ←. sensor; ltl ←. reset;
   leftturn ←. sensor; leftturn ←. reset;
   cars ←. sensor; cars ←. reset
      SFD begin
      input Boolean array sensor [1:5];
      input Boolean reset, light;
      output Boolean ltl, leftturn;
      output integer cars;
      integer i,j;
      Boolean lastleftlight;
      la: time begin
         if reset then
            begin
            leftturn ← false;
            ltl ← false;
            cars ← 0;
            end;
         for i←1,2 do if sensor [i] then
            cars ← cars + 1;
         for i←3,4 do if sensor [i] then
            cars ← cars - 1;
         if sensor [5] then leftturn ← ltl ← true;
         if leftturn ∧ lastleftlight then
            begin
            leftturn ← false;
            cars ← cars - 1;
            end;
         lastleftlight ← light; go to la;
         end of time block;
      end SFD ALGOL description
```

declaration of inputs, outputs, and interconnectors for the traffic control module

— component declaration

declaration of subsystem inputs and outputs

declaration of immediate dependencies

Note #1

updates count of cars waiting in each leg, indicates if left-turn lane exists and stores current values of left-turn lights

Note #1: This redeclaration of inputs and outputs allows the sublanguage compiler to operate independently of the SODAS compiler. In the future this redundancy will be eliminated.

Figure 7—SODAS description of traffic control module at level two

SODAS Description	Comments
subsystem counter sensor [1] ← sensor [1] ∧ sensor [2] ← sensor [2] ∧ sensor [3] ← sensor [3] ∧ sensor [4] ← sensor [4] ∧ sensor [5] ← sensor [5] ∧ reset ← reset ∧ light ← light [3] ∧ ltl → L[1] ∧ leftturn → leftturn [1] ∧ cars → cars [1] ;	counter for leg 1 - reference to component defined above specification of connections to interconnectors
subsystem counter sensor [1] ← sensor [6] ∧ sensor [2] ← sensor [7] ∧ sensor [3] ← sensor [8] ∧ sensor [4] ← sensor [9] ∧ sensor [5] ← sensor [10] ∧ reset ← reset ∧ light ← light [6] ∧ ltl → L[2] ∧ leftturn → leftturn [2] ∧ cars → cars [2] ;	counter for leg 2
subsystem counter sensor [1] ← sensor [11] ∧ sensor [2] ← sensor [12] ∧ sensor [3] ← sensor [13] ∧ sensor [4] ← sensor [14] ∧ sensor [5] ← sensor [15] ∧ reset ← reset ∧ light ← light [9] ∧ ltl → L[3] ∧ leftturn → leftturn [3] ∧ cars → cars [3] ;	counter for leg 3
subsystem counter sensor [1] ← sensor [16] ∧ sensor [2] ← sensor [17] ∧ sensor [3] ← sensor [18] ∧ sensor [4] ← sensor [19] ∧ sensor [5] ← sensor [20] ∧ reset ← reset ∧ light ← light [12] ∧ ltl → L[4] ∧ leftturn → leftturn [4] ∧ cars → cars [4] ;	counter for leg 4

SODAS Description	Comments
subsystem	— joiner
input Boolean array L[1:4];	declaration of: — inputs
output Boolean ltl	— outputs
L[1] ← L[1] ∧ L2] ← L[2] ∧ L[3] ← L[3] ∧ L[4] ← L[4] ∧ ltl → ltl	— connections
ltl ↩ L	— immediate dependencies
BOOLE begin input L[1:4]; element ltl; gate ltl type or inputs L[1], L[2], L[3], L[4]; outputs ltl; end BOOLE description of joiner;	BOOLE description of "or" gate
subsystem	— control unit
input Boolean array outside light [1:12]; input integer array cars [1:4]; input Boolean loc, coc, reset, ltl; input integer tmax, tmin, ttot, initialmode;	input declarations
output Boolean array light [1:12]	output declaration
outsidelight[1]←outsidelight[1]∧light[1]→light[1]∧ outsidelight[2]←outsidelight[2]∧light[2]→light[2]∧ outsidelight[3]←outsidelight[3]∧light[3]→light[3]∧ outsidelight[4]←outsidelight[4]∧light[4]→light[4]∧ outsidelight[5]←outsidelight[5]∧light[5]→light[5]∧ outsidelight[6]←outsidelight[6]∧light[6]→light[6]∧ outsidelight[7]←outsidelight[7]∧light[7]→light[7]∧ outsidelight[8]←outsidelight[8]∧light[8]→light[8]∧ outsidelight[9]←outsidelight[9]∧light[9]→light[9]∧ outsidelight[10]←outsidelight[10]∧light[10]→light[10]∧ outsidelight[11]←outsidelight[11]∧light[11]→light[11]∧ outsidelight[12]←outsidelight[12]∧light[12]→light[12]∧ loc←loc∧coc←coc∧reset←reset∧ltl←ltl∧ tmax←tmax∧tmin←tmin∧ttot←cycletime∧ initialmode←initialmode∧cars[1]←cars[1]∧ cars[2]←cars[2]∧cars[3]←cars[3]∧cars[4]←cars[4]	connections to inter-connectors
light ↩ reset; light ↩ loc; light ↩ coc; light ↩ outsidelight; light ↩ ltl; light ↩ tmax; light ↩ tmin; light ↩ cycletime; light ↩ initialmode; light ↩ cars	declaration of immediate dependencies

SODAS Description

```
SFD begin
   comment control unit subsystem;
   input Boolean array outsidelight [1:12];
   input Boolean loc, coc, reset, ltl;
   input integer array cars [1:4];
   input integer tmax, tmin, ttot, initialmode;
   output Boolean array light [1:12];
   integer mode, tim, ctmax, ctmin, cttot;
   real factor;

integer procedure newmode;
   begin
   integer i, high, nscars, ewcars, tcars;
   nscars←cars[1] + cars[3];
   ewcars←cars[2] + cars[4];
   tcars←ewcars + nscars;
   if tcars=0 then tcars←1;
   if ¬ ltl then
      begin
      high←0;
      for i←1,2,3,4 do
         if i ¬ = mode ∧ (cars[i]>0 ∨ loc) then
            begin
            if cars [i]>high then
               begin
               high←cars[i]; newmode←i;
               end;
            end;
         factor←high/tcars;
      end else
      begin
      if mode ¬ =5∧mode ¬ =7∧(leftturn[1]∨leftturn[3]) then
         begin newmode←7; factor← 5*nscars/tcars;end else
      if mode ¬ =5∧mode ¬ =7∧(leftturn[2]∨leftturn[4]) then
         begin newmode←5; factor← 5*ewcars/tcars;end else
      if mode ¬ =8 ∧ (nscars>ewcars) then
         begin newmode←8; factor←nscars/tcars;end else
         begin newmode←6; factor←ewcars/tcars;end;
      end;
      end of new mode procedure;

integer procedure cycletime;
   if mode = 5 ∨ mode = 7 then cycletime ← tmin
   else if ¬ loc then cycletime ← tmax
   else cycletime ← factor*cttot;
```

Comments

```
control unit
now has more
inputs and
fewer internal
variables

selects new
mode to maximize
the number of
cars allowed
to move

determines the
time to be spent
in the new mode
```

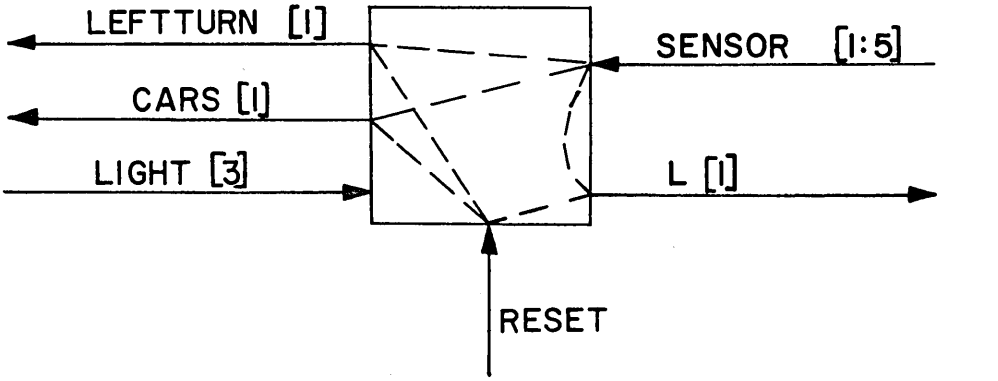
SODAS Description	Comments

`procedure initializecontroller;`
`  begin`
`  integer i;`
`  for i←1 step 1 until 12 do light[i]←false;`
`  mode ← initialmode;`
`  tim←tmax; factor←1;`
`  ctmax←tmax; ctmin←tmin;`
`  cttot← if loc then ttot else 1800;`
`  ltl←false;`
`  end initialization of the controller;`
	initializes internal variables and outputs

`procedure setlights (a,b); Boolean array a; integer b;`
`  begin`
`  integer i;`
`  for i←1 step 1 until 12 do a[i]←false;`
`  if b=5 then a[ 1]←a[ 6]←a[ 7]←a[12]←true else`
`  if b=6 then a[ 4]←a[ 5]←a[10]←a[11]←true else`
`  if b=7 then a[ 4]←a[ 3]←a[10]←a[ 9]←true else`
`  if b=8 then a[ 1]←a[ 2]←a[ 7]←a[ 8]←true else`
`  if b=2 then a[ 1]←a[ 4]←a[ 5]←a[ 6]←true else`
`  if b=4 then a[ 7]←a[10]←a[11]←a[12]←true else`
`  if b=1 then a[ 1]←a[10]←a[ 2]←a[ 3]←true else`
`  if b=3 then a[ 4]←a[ 7]←a[ 8]←a[ 9]←true;`
`  end setlights;`
	sets traffic lights, given desired mode

`continue:`
`  time begin`
`  if reset then initializecontroller;`
`  if coc then`
`  begin integer i;`
`  for i←1 step 1 until 12 do`
`  light [i] ← outsidelight[i];`
`  go to done;`
`  end;`
`  if loc then cttot ← ttot else cttot ← 1800;`
`  tim ← tim +1;`
`  if tim>ctmax then`
`  begin`
`  tim ← 1;`
`  mode ← newmode;`
`  ctmax ← cycletime;`
`  setlights (light, mode):`
`  end;`
`  done: go to continue;`
`  end of time block;`
`end of SFD ALGOL description of control unit`
	this is the same control algorithm discussed in Figure , except cars is now an input and not an internal variable

`end SODAS description of traffic control module`	



LEG 1 COUNTER AT SECOND LEVEL OF DESCRIPTION

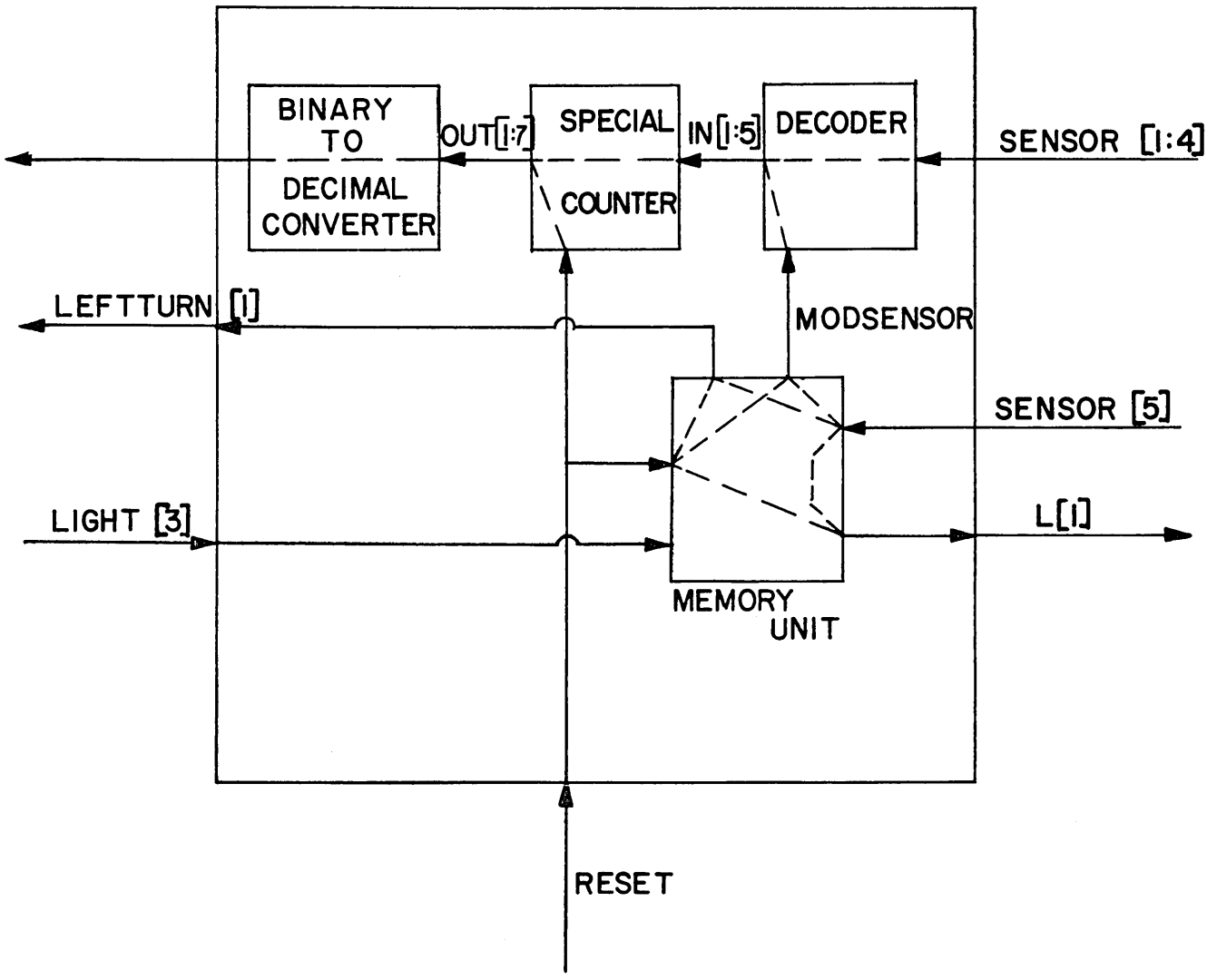


Figure 8—Leg 1 counter at third level of description

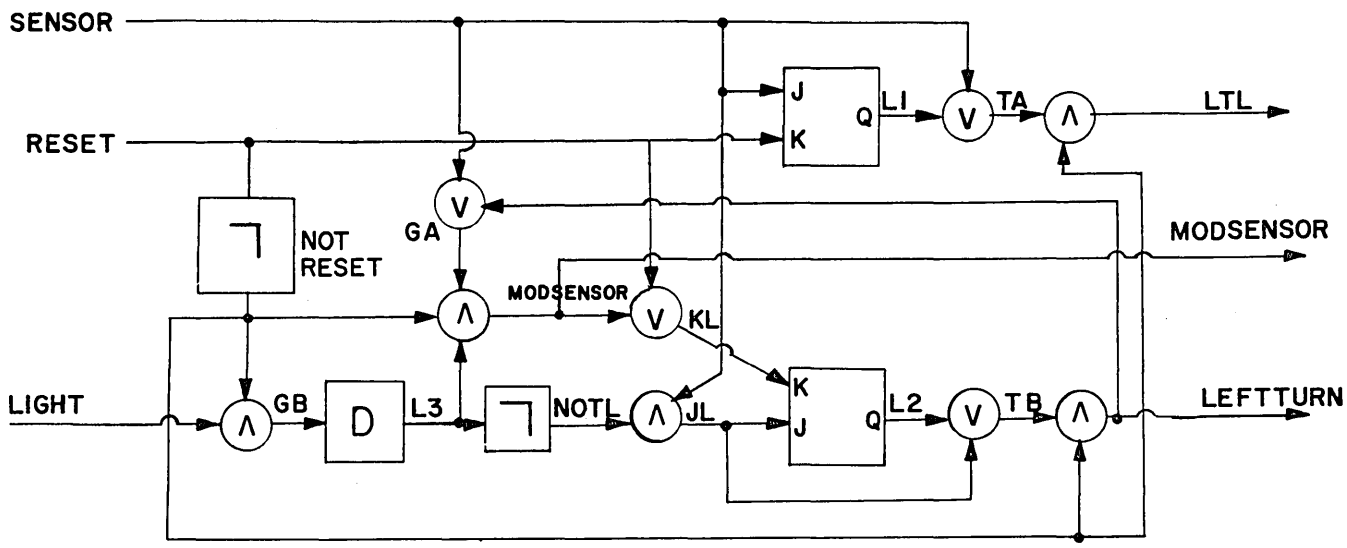


Figure 9—Logic diagram for memory unit

the reader should note the strong correspondence between block diagrams and the structure of the SODAS program.

We consider this correspondence one of the most important features of SODAS and its sub-language BOOLE. There is a simple (mechanical) process to go from a block diagram to a SODAS description of a system, or from a logic diagram to a BOOLE description of a network. SODAS does not restrict or influence the designer's way of thinking—except possibly to encourage him to be more precise in his definitions.

CONCLUSIONS

We have presented a methodology for systems design, and a language that we feel is a significant aid to systems designers. The aid that SODAS provides is probably best indicated by the phrase **S**tructure **O**riented **D**escription **A**nd **S**imulation. SODAS is unique in its ability to describe systems as structures of components that operate in parallel. It is also unique in its ability to simulate such parallel structures correctly. We feel that these two features are almost indispensable in a software system to aid in the systems design process. As evidence we offer the example of the traffic control module; although the system is a small and simple one by today's standards, SODAS appears to be a substantial aid to the design and description of the example. Interested readers may wish to compare it with the description of student solutions to the same problem.[9]

As an example of one of the incidental benefits of such a system, consider the problem of grading students (or evaluating the work of members of a professional design team). In either situation it is extremely difficult to evaluate the work of the individual team members when part of the system fails or is not completed. It is extremely difficult to determine just who is at fault, and, more important, to evaluate the work of the remaining team members. With SODAS, the specification of part of a system can be substituted for the actual design in determining if the remainder of the system functions correctly. Further, since there are precise and testable specifications, it is possible to determine which components do not meet those specifications. It appears to us that this will be a valuable tool for the managers of system design projects.

Our work on the example has indicated that the use of SODAS forces a discipline upon the system designer that could greatly improve the quality of his work and reduce the time needed to complete a project. We have found that this discipline carries over to design problems in which, for one reason or another, SODAS is not used (e.g., the design of the SODAS system itself).

REFERENCES

1 D L PARNAS
Sequential equivalents of parallel processes
Center for the Study of Information Processing Carnegie Institute of Technology Pittsburgh Pennsylvania

2 N WIRTH H WEBER
Euler: A generalization of ALGOL and its formal definition
Part I *CACM* 9 pp 13-23 January 1966

3 *Ibid*
CS 20 Technical Report Computer Science Department Stanford University Stanford California

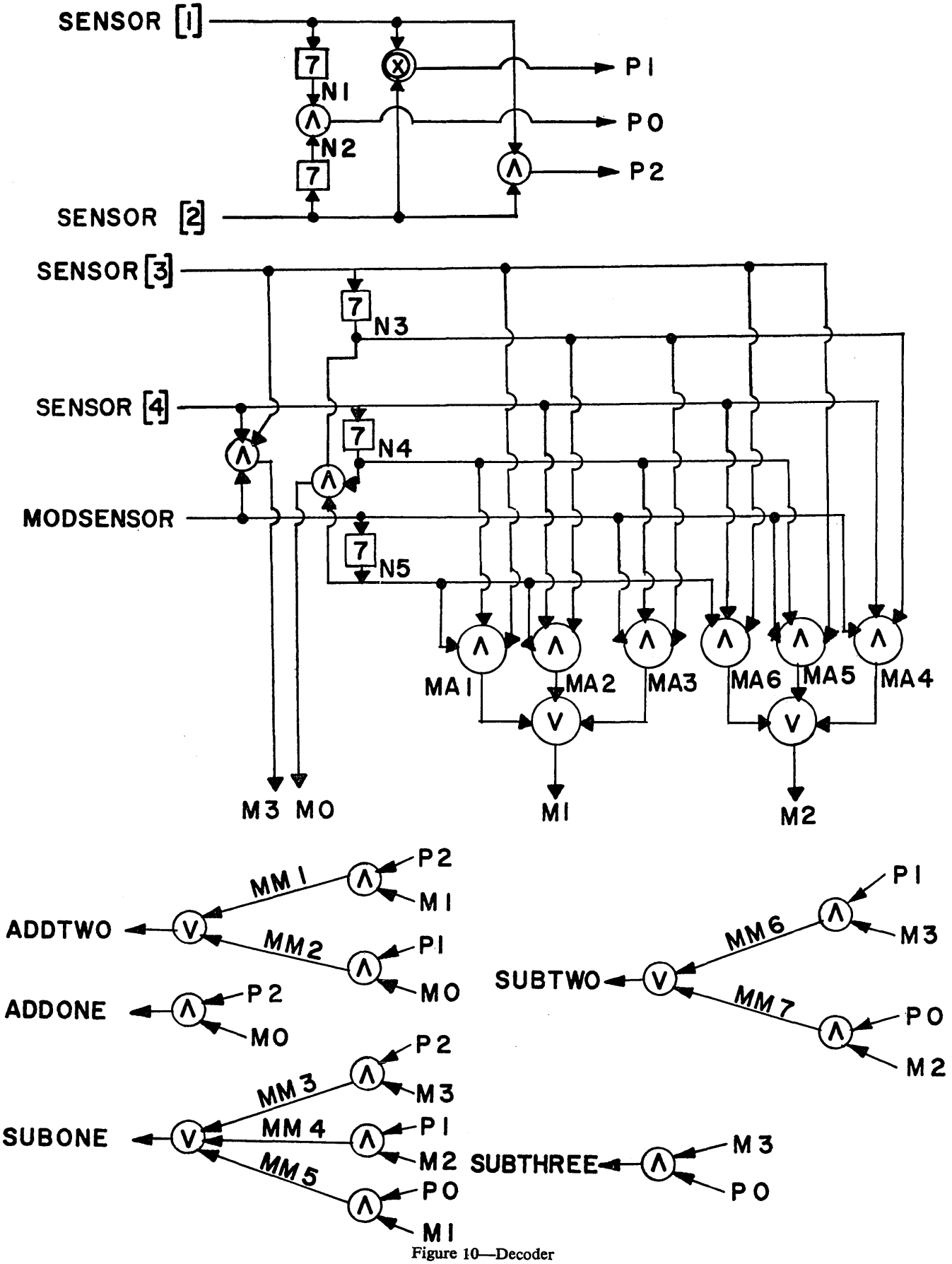


Figure 10—Decoder

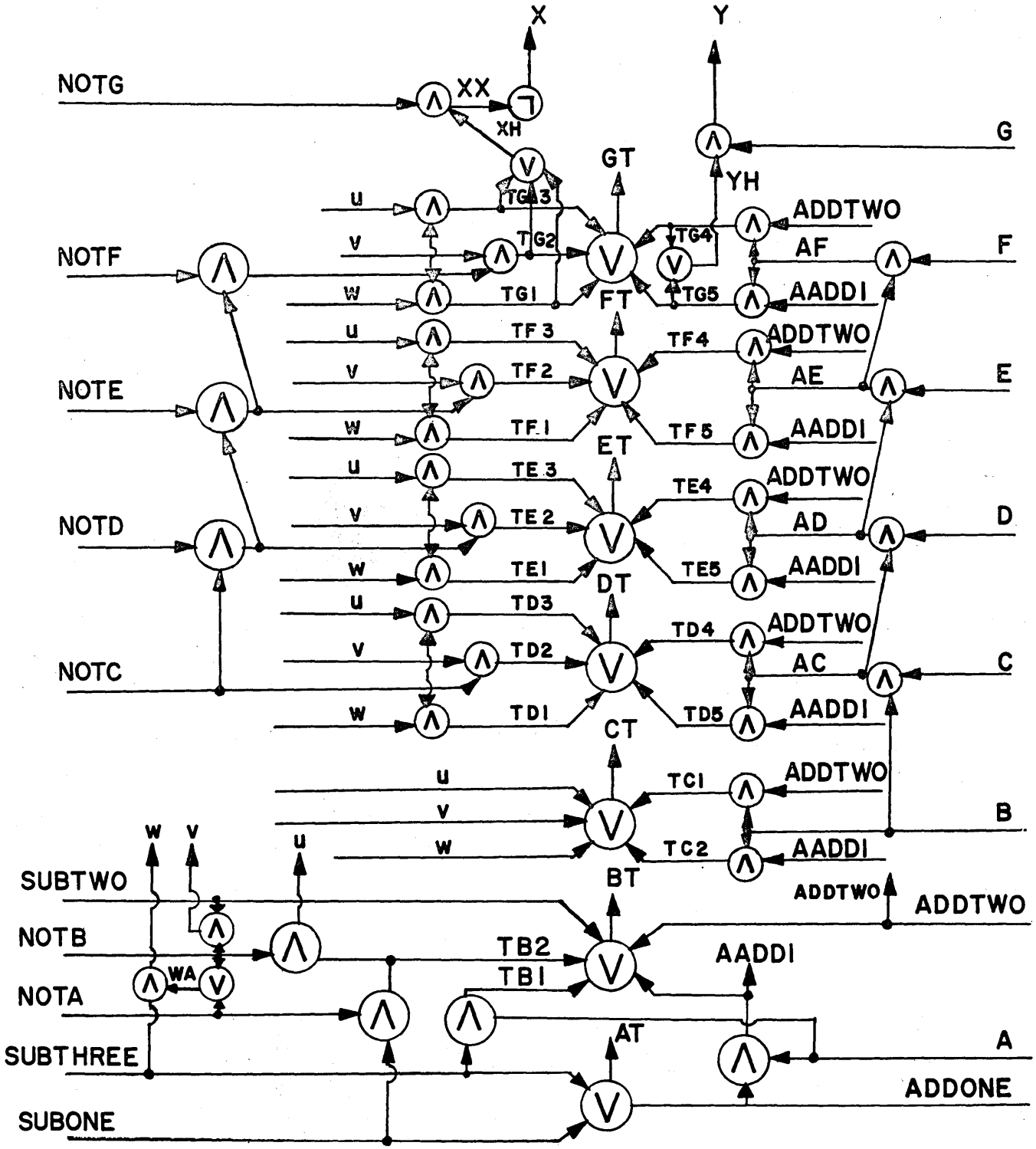


Figure 11—Gaiting for special counter

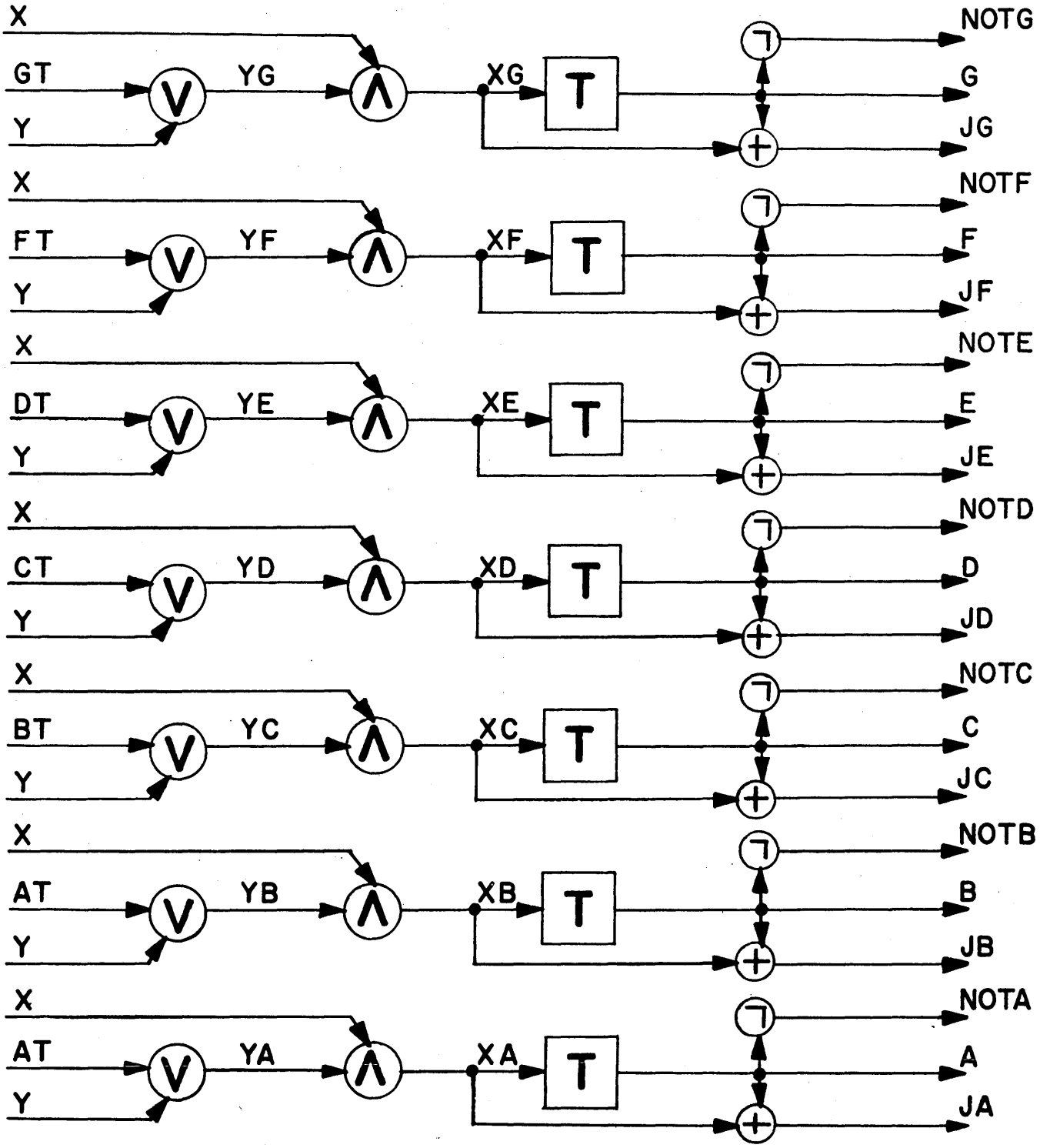


Figure 12—Memory for special counter

SODAS Description	Comments
subsystem	– memory unit
input Boolean sensor, light, reset;	declaration of inputs and outputs
output Boolean ltl, leftturn, modsensor	
sensor ← sensor [5] ∧ light ← light [3] ∧	declaration of interconnections
reset ← reset ∧ ltl → ltl ∧	
leftturn → leftturn [1] ∧	
modsensor → modsensor	
ltl ↔ reset;	declaration of immediate dependencies
leftturn ↔ reset;	
ltl ↔ sensor; leftturn ↔ sensor;	
modsensor ↔ sensor;	
modsensor ↔ reset	
BOOLE begin	Note #1
input sensor, light, reset;	
element notreset, ga, gb, jk, jl, ta, tb,	declarations of circuit elements and their interconnections
L[1:3], ltl, leftturn, notl, modsensor;	
gate notreset type not inputs resets;	
gate ga type and inputs notreset, light;	
gate ga type or inputs sensor, leftturn;	
gate modsensor type and inputs ga, L3, notreset;	
gate ta type or inputs sensor, L1;	
gate tb type or inputs jl, L3;	
gate ltl type and inputs ta, notreset;	
gate leftturn type and inputs tb, notreset;	
gate jl type and inputs sensor, notl;	
gate notl type not inputs L3;	
gate kl type or inputs modsensor, reset;	
flip-flop L1 type jk inputs sensor, reset;	
flip-flop L2 type jk inputs jl, kl;	
flip-flop L3 type d inputs gb;	
outputs ltl, leftturn, modsensor;	Note #1
end BOOLE description of memory unit;	

Figure 13—Boole description memory unit

4 P NAUR (ed)
Report on the algorithmic language ALGOL 60 (revised)
CACM 6 pp 1-17 January 1963

5 D E KNUTH J L McNELY
SOL—A symbolic language for general purpose systems simulation
IEEE Transactions on Electronic Computers August 1964
D E KNUTH J L McNELY
A formal definition of SOL
IEEE Transactions on Electronic Computers August 1964

6 D L PARNAS L C RICHARDSON W H KOHL
Preliminary version—an introduction to Boole-66
Unpublished manual available from Computation Center Carnegie Institute of Technology

7 D L PARNAS
A language for describing the functions of synchronous systems
Comm ACM February 1966

8 J FIERST (ed)
ALGOL 20 language manual
Computation Center Carnegie Institute of Technology Pittsburgh Pennsylvania

9 J C STRAUSS D L PARNAS Y WALLACH R W SNELSIRE
A design emphasis problem solving experience
Department of Electrical Engineering Carnegie Institute of Technology

10 IBM Corporation White Plains New York Introduction to General Purpose Systems Simulator III

11 H M MARKOWITZ B HAUSNER H W KARR
SIMSCRIPT a simulation programming language
Prentice Hall Inc Englewood Cliffs New Jersey 1963

12 O J DAHL K NYGAARD
SIMULA—An ALGOL based simulation language
Comm ACM pp 670-678 September 1966

Requirements for a shared data processing system for hospitals

by JOHN P. BODKIN
Minnesota Hospital Service Association
St. Paul, Minnesota

INTRODUCTION

The Service to Minnesota Hospitals program is one in which Minnesota Blue Cross, together with participating hospitals, have joined in making available the best possible EDP services to hospitals in this area.

Consideration of this program began in 1963 when a feasibility study commenced in which Blue Cross and four hospitals in Minneapolis participated with approvals from the Twin City Hospital Association and the Minnesota Hospital Association. This study was concluded in mid-1964 and the results were incorporated in the publication entitled "Cooperative EDP for Minnesota Hospitals" released to all hospitals in the State in September, 1964. The conclusions reached by the Study Committee were that a cooperative EDP system for Minnesota hospitals in which centralized computer facilities would be installed at Minnesota Blue Cross is definitely feasible and would result in significant benefits to the hospitals served. It was further recommended that the cooperative program should result in the eventual installation of an integrated system of EDP services for all feasible applications in hospitals but that the initial applications to be developed should be referred to as "Stage I" and should consist of:

- Patient Accounting.
- Payroll and Personnel Record Accounting. (Employee Information System)
- Inventory Control and Purchasing.
- Accounts Payable Accounting.
- Property Ledger Accounting.
- Preventive Maintenance Scheduling.
- General Ledger Accounting.

The principal advantages and disadvantages of a cooperative as compared to individual hospital EDP effort were concluded to be the following.

Advantages

The cooperative approach to EDP provides greater potential benefits to hospital operation for less cost. Some of the individual factors which justify this observation are as follows:

1. The combination of equipment and staff under a cooperative system can perform greater services due to *Greater Capability of Equipment*—a cooperative system can afford the computing and peripheral equipment to most effectively perform all hospital functions that should benefit from EDP. Technological obsolescence can be more effectively coped with as new and better equipment becomes available since there is a broad base over which to share any replacement costs.
   Skilled Systems, programming and operating staff can be retained. The highest degree of data processing skills can be brought to bear in the development and operation of a cooperative system. Desirable specialization of functions can be achieved. Retention of key individuals in small individual systems can become extremely difficult due to the competition that exists.
   A broad base of analysis is made available through the cumulative hospital experience of numerous hospitals.
   Contributions in Consultation as well as direct effort from outside sources can be very significant particularly in a cooperative approach. These outside contacts include equipment manufacturers, consultants, other hospital and hospital-related institutions, etc. These contributions will be particularly valuable in the development and realization of long-range systems planning.
   More Operating Systems for each hospital will result in a shorter period of time due to the ability

of a relatively large staff to simultaneously plan for and implement several areas of EDP services.

2. *Initial Investment*—Under the cooperative approach, the initial investment is substantially reduced since all costs are recovered through monthly service charges that commence only when a system becomes operational for that hospital.
   Elements of this initial investment expense consist of such items as:
   Salaries for systems analysis, programming, proect director, training of operating staff, etc.
   Equipment Rental for any period of time during which the computer is on rent and not yet productive (or fully productive).
   Construction Costs for a data processing room including necessary temperature and humidity control, special electrical power, etc.
   Supplies—particularly magnetic storage (tapes or other type) which are a costly item of initial expense.
3. *Supporting Staff*—The installation and operation of an EDP system requires the efforts of a skilled and well-managed staff. In an individual as well as in a cooperative system, all these skills must be procured and retained. The personnel expense for this staff is significant. In a cooperative approach, however, the cost per hospital through the monthly service charges is drastically reduced.
4. *Equipment Rental*—It is fundamental that potential processing cost per unit is cheaper as the size of the computer is increased. Therefore, substantial ultimate benefit to participating hospitals due to the fact that larger, less expensive per unit of processing equipment is utilized.

Disadvantages

The cooperative approach inherently requires willingness to compromise individual hospital interests where necessary to achieve the benefits that can be realized. Some elements of this compromise are:

1. *Systems Design*—The precise design of operating systems can be made flexible to include numerous options which will accommodate the desires of one or more hospitals. This, however, is not always economically practical.
2. *Planning*—Planning for future systems must be done cooperatively.
3. *Priorities*—Decisions as to the sequence of applications to be installed as operating systems will have to be worked out on a basis of the aggregate benefit to all participating hospitals. The same would be true with respect to selection of future applications for further research and implementation.

Hospital data processing council

To coordinate the activities of the EDP program, a Hospital Data Processing Council has been established. This Council consists of administrative representation from each hospital participating in the program. The purposes of this Council are to receive recommendations from the Blue Cross staff and to make all final decisions regarding the characteristics of the systems that are to be offered.

Participating hospitals

Data processing service by Blue Cross to Minnesota hospitals commenced in 1955 when the first hospital, St. Barnabas of Minneapolis, transferred its Payroll processing to Blue Cross. Since that time, this Payroll service has consistently increased until currently there are over 30 hospitals in Minnesota whose payroll is being processed by Blue Cross. This system has undergone many changes since 1955 and is now a completely computerized operation. This service handles the payroll for approximately 22,500 hospital employees. A second service that was added was Discharged Accounts Receivable processing in 1963.

The above services of Payroll and Discharged Accounts Receivable were not planned on a formally-organized basis with hospitals and will be eventually merged into the cooperative program otherwise described. Those hospitals which have to date elected to participate in the cooperative EDP program and whose activities are governed by the Hospital Data Processing Council are as follows.

Minneapolis
Abbott Hospital
Lutheran Deaconess Hospital
Eitel Hospital
Mount Sinai Hospital
Northwestern Hospital
St. Barnabas Hospital
The Swedish Hospital
The University of Minnesota Hospitals

St. Paul
St. John's Hospital

St. Cloud
St. Cloud Hospital

By July, 1967, the Patient Accounting System will be installed in all participating hospitals. All other Stage I Systems will either be installed, or available for installation, by the end of 1967.

Financial policies

Charges by Blue Cross for hospitals serviced under the cooperative EDP program are based on actual costs of operation, including amortization of necessary development expenses. Cost accounting methods are used to assure that hospitals are not charged for the services beyond the actual costs of operation, nor will the operation be subsidized by income from Blue Cross subscriber payments.

Computer center equipment

The computer center at Blue Cross currently consists of two Honeywell Series 200 computers. One computer, an H-200, is primarily a communication processor and is connected by communication lines to each hospital.

As of May, 1967, the configuration of this system was as follows:

H-200 Central Processor—28K (characters) memory.
1—Tape Control Unit.
2—20kc Magnetic Tape Drives.
2—High Speed Random Access Drums (2.6 million characters of storage-each drum).
1—High-speed Printer.
1—Console Printer.

This system is the interface between the hospitals and the accounting systems that are run on another computer.

All input for the various accounting systems is prepared in the hospital and transmitted via communication lines to this system. Batch total reports, error reports, census reports, and other low volume reports are transmitted back to the hospital via this computer.

There is also a once a day update of the random access drum, where summary bills for in-hospital patients are stored. Summary bills and other patient data are printed in the hospital on demand.

In addition to handling traffic to and from the hospitals, the system also drives a high-speed printer utilizing print image tapes from other processors.

At the present time, all devices connected to this system operate at their maximum speed, simultaneously. We do not anticipate a noticeable slowdown of individual peripheral devices until after thirty (30) communication lines have been connected to the system. This slowdown would then only occur in a worst-case condition.

The other computer used for these accounting systems is a Honeywell 1200 equipped as follows:

2—Tape Control Units.
4—44kc Magnetic Tape Drives.
4—67kc Magnetic Tape Drives.
1—High-speed printer.
1—High-speed card reader.
1—Console Printer.

All processing for the accounting systems is accomplished on this system. Quite frequently this system operates in the foreground/background mode. In other words, card-to-tape, and tape-to-printer operations occur simultaneously with, but independent of, other processing.

As additional hospitals and/or services are added to the cooperative program beyond the capacities of the installed units, the Series 200, as well as equipment offered by other manufacturers, offers built-in growth potential so that all types of services desired by participating hospitals can actually be performed by the computer center.

Mechanisms for adding capacity for the present Series 200 systems consist of the following:

- Additional Series 200 computers.
- Successively higher speed central processors which offer growth without program translation thru higher speed execution of instructions, and increased capability in the area of multi-programming.
- Additional internal memory. Maximum memory size of a single Series 200 processor is over one (1) million characters.
- Random access storage. Several types of random access devices are available.

Communications and terminal units

All accounting systems are on-line systems requiring data communications lines and terminal keying and printing units to be located in the hospital. Specific information relative to these devices is as follows.

ACS-35

This is a combination keying, printing and paper tape reading/punching device provided by the Northwestern Bell Telephone Co. and manufactured by the Teletype Corporation. It is directly connected to the H-200 communication computer via communication lines.

Basically, this device is an ASR-35. It has been modified to include a second paper tape reader and a stepping circuit. The purpose of this modification was to provide format control when keying data for the accounting systems. This device operates at a speed of ten (10) characters per second.

Friden add punch

This is a free-standing paper tape punching device operated by a ten-digit keyboard for use in punching

numeric only data. The device includes a check digit verifier which will automatically detect errors in keying when they occur, and before they are recorded in the punched paper tape. The paper tape created by this device is transmitted to the computer center utilizing the ACS-35.

The above units are the initial units specified for operation with the various accounting systems. Improvements will be made in this area from time to time as the situation warrants it. We expect these changes will be in the following areas:

- High-speed transmission of paper tape.
- Faster character printing capability.
- Line Printers.
- Cathode ray tubes for use as input and temporary display devices.

What systems shall a cooperative group implement first?

In any cooperative venture into Hospital Data Processing, several factors must be taken into consideration before a cooperative group can make a decision as to what area of the hospital is to receive the benefits of data processing first. Some of the factors are:

Objectives of the cooperative group

Long-range and short-range objectives of the cooperative group must be outlined. Following are some questions that should be answered.

- Do we desire to develop or acquire a standard Hospital Information System that would be installed in all hospitals, or, on the other hand, do we merely wish to share a central computer and develop our own systems, designed to an individual hospital's specifications?
- To what extent do we desire standardization of systems and procedures?
- To what extent do we wish to share or compare information concerning patients, employees, supplies, services, etc.?
- What hospitals in the community or the state will these services be offered to, and what will be the method of input—On-line or off-line?

Source and amount of funds available

Are they:

- A Government or private grant?
- Hospital supplied?
- A loan?
- Must they be repaid or merely accounted for?

Data processing staff

- What are their skills and backgrounds?
- What are their abilities and knowledge of the task at hand in the areas of:
  Systems analysis?
  Systems design?
  Computer and related hardware?
  Computer software?
  Hospitals?

Equipment suppliers

- What kind and how much support are the equipment suppliers willing to give?
- What application packages or capabilities, that will assist you in attaining your objectives, do they have or will commit themselves to supplying?

Computer hardware and software

- Is the hardware and software, that will enable you to achieve your systems objectives, available on the market?
- Can it be delivered in time to meet your implementtation plan?
- Can you afford it?

Accounting systems first in Minnesota

After taking all of the above factors, and many others, into consideration, the Study Committee concluded that a cooperative EDP System for Minnesota hospitals utilizing a centralized computer facility was definitely feasible, desirable, and would result in significant benefits to hospitals served. This cooperative system would provide EDP services for substantially less cost than would be incurred for comparable services if hospitals individually installed computers. The services rendered through cooperative data processing would tend to be superior to services which could be rendered by individual hospital computers due to greater computing capacity and the potential for economical growth.

The recruiting and retention of a skilled systems, programming, and operating staff would be easier and more economical on a group basis. Substantial duplication of effort in research (system analysis), systems design, and programming activities would be avoided as compared to several hospitals doing these things separately.

It was determined that there are many types of data processing services which could be provided hospitals by such a cooperative system. Some of the potential functions were currently being performed in certain hospitals, such as patient accounting, inventory control, payroll, etc. There are many other potential applications, however that, for the most part, have not as yet been defined, and developed.

There is great interest today in EDP as an aid to the clinical practice of medicine. This could include, for

example, the handling by electronic means of all or much of the information in the patient's chart. There can be no question that once clinical applications are perfected and adopted by the medical and professional hospital staffs, they will be of substantial benefits to health care.

There is, however, widespread consensus that hospitals must initially install those EDP applications that are practical, proven, easily defined, and understood, and do not add significant cost to patient care. For these reasons, we chose to implement those accounting applications previously referred to as "Stage I Systems."

In retrospect, both the member hospitals and Minnesota Hospital Service Association feel that this was a very sound decision. We feel that the experience gained by the data processing staff and the administration of the hospitals will be invaluable in the development of future systems or extensions and improvement of the initial effort.

Future systems development

During the remainder of 1967, a detailed systems analysis will be made in two areas of the hospital—the Clinical Laboratory and the Pharmacy.

In a recently completed survey, four major problem areas in the Laboratory were identified that have potential EDP solutions. They are:

1. *Test ordering*—Can be made more accurate and much faster. Can be used to prepare lab worksheet and specimen collection schedule. Should include a better, more efficient means of specimen identification. A by-product of test ordering would be automatic preparation of charge data for the Patient Accounting System.

2. *Test Reporting*—Can be faster. Can have a better form a display than "shingles," and can be made available for instant display in areas such as the Doctors' Lounge, lab, or nursing station.

3. *Quality Control*—Is presently very time-consuming and subject to manual inaccuracies. The personnel interviewed would like improved quality control procedures as part of any new system.

4. *On-Line Monitoring* — Automated equipment, which most of the laboratories have installed or ordered, allows many more tests to be performed. Results must still be monitored and recorded. If these devices were on-line to a computer, monitoring and recording would be automatic.

Characteristics of such a laboratory system would be as follows.

Laboratory requisitions would be entered into the system from the physician's order by the nursing staff, or directly by the physician. As a result of this action, the system would produce the following:

1. Specimen collection schedules.
2. Specimen labels.
3. Laboratory work schedule.
4. A charge to be posted to the patient's account.
5. Updated file of ordered but not completed tests.

When specimens are collected and delivered to the laboratory, they will have the following positive identification:

1. Accession number.
2. Patient number and/or name.
3. Test number and/or name.
4. Date.
5. Time required.

It would be desirable if a portion of the label were in machine-readable format if the test is to be run on automated test equipment.

When a series of tests are set up, either manually or on automatic testing equipment, basically the same procedure is used—the test is calibrated by running a control specimen or a series of controls. Controls may be interspersed to verify continued accuracy of test results. The verification of control results should be performed by the computer, either manual entry of raw results or automatic entry via on-line monitoring.

During a run with an on-line automatic testing machine, results may be adjusted by the computer for drift and specimen interaction as detected by control specimen tests. As tests are completed and filed in the central computer, a report, together with previous tests for the patient, would be made available in the laboratory, the nursing station, and any other location required by the medical staff.

With all test results available on-line, it will be possible to recall a patient profile at any time. Display of the profile can be a permanent hard copy printed by the character printer or temporary display on the cathode ray tube. The profile would display the results of all completed tests and, in addition, tests that have been ordered *but not completed.*

Completed tests will be automatically audited against tests scheduled; an exception reporting system for the laboratory will assure that all ordered collections have been made and that the tests have, in fact, been completed. We feel that the benefits of such a system would be:

1. Reduced clerical effort in the laboratory.
2. Reduced clerical effort in the nursing station.
3. Increased accuracy in test ordering.
4. Better quality control.
5. Improved reporting and display of results.
6. Lost or misplaced test results will be detected and reported.

7. Fewer medical technologists will be able to perform more tests.
8. The ability to perform admission screening and multi-testing economically and rapidly benefits the laboratory, the medical staff, and the patient.

Pharmacy system

Listed below are some of the reasons that we feel that data processing can play a major role in the Pharmacy and medication administration.

- Studies have shown that 30-40% of physicians' orders are for medications.
- Much of the pharmacist's time is spent in clerical work—ordering, stocking, taking inventory, typing prescription labels.
- Medication charges lack accuracy because of clerical work involved.
- Errors due to poor communication and control directly affect patient care.
- A large amount of nursing time is spent on paper work associated with medications.

Our present concepts are that a Pharmacy System would be developed in three stages. These three stages, as well as the benefits to be derived from each, are listed below.

Stage one

Development of a computer-based inventory and formulary catalogue.

Benefits:

- Reduced clerical effort in the pharmacy.
- Improved purchasing including group buying.
- Formulary listing by therapeutic usage makes generic dispensing feasible.
- Drug location index assists inventory and order filling.

Stage two

Integration with Patient Accounting.

Benefits:

- Changes based upon drug cost to both inpatient and outpatient prescriptions.
- Patient drug profiles may be prepared off-line for medical records, probably for medication administration.
- Statistics including drug usage and adverse reactions may be a by-product.

Operating reports may be prepared automatically.

Stage three

On-line, real-time medication administration system.

Benefits:

- On-line drug profiles available in pharmacy, at any other locations.
- On-line medication schedules.
- Automatic stop or hold orders.
- Computer preparation of prescription labels.

Hardware requirements

If it is determined that it is feasible to implement the above concepts in total, we feel that a real-time computer with the following capabilities will be a requirement in the hospital:

1. Communication switching—intrahospital and hospital to the computer center.
2. Analog digital capabilites.
3. Scientific computing capabilites.
4. In-hospital real-time computing capabilites.
5. High-speed random access for storage of programs and small data files.
6. The system must be capable of communicating with a mutiplicity of terminal devices such as cathode ray tubes, teletype printers, and the central computer center.

The system must be capable of being expanded so that other departments in the hospital, such as admitting, dietary, X-ray, may also be served. The computer center hardware must also be upgraded so that patient files are on-line and available at all times.

Use of displays with packaged statistical programs

by *W. J. DIXON*
University of California
Los Angeles, California

During the past few years there has been a great increase in the use of packaged statistical programs. These programs are prepared in a general form. For example, a regression program will allow the user to specify:

- the number of variables being introduced as a data set
- the number of cases
- the choice from this data set of the dependent variable
- the choice of some subset of the input variables to be the independent variables
- the type of input (cards vs. tapes, etc.)
- the transformation of any variable in the data set
- the construction of new variables for some stated function of the input variables
- the stepwise computation of regression function
- the priority with which variables may be considered for introduction into the regression equation
- the plotting of residuals vs. various input variables.

The output provided includes means, standard deviation, correlation, and at each step of the regression process regression coefficients, partial correlation, and an analysis of variance table for the regression.

At the end of the process a list of residuals and the plots mentioned above are provided.

A regression program of this generality will enable a user to compute a wide variety of problems, as well as the standard regression problem, e.g., analysis of variance, contrasts either orthogonal or dependent, analysis of covariance, discrimination, etc.

A library of statistical programs such as the BMD series developed at UCLA gives the user a selection of several different regression packages as well as a variety of data screening, analysis of variance, multivariate analysis and other statistical procedures, all providing a considerable degree of flexibility to cover a wide variety of problems.

In biomedical applications, as well as in other fields, studies with large amounts of data are analyzed sequentially. Various analyses are carried out to choose among various possible dependent or outcome variables and to relate these variables to the many independent variables. Analyses are made to screen the data for accuracy, including search for outliers, etc. The data are examined for linearity, homogeneous covariance matrices, etc. The investigator can be assisted by the computer during this phase, but he must still participate actively in directing the process.

If these programs are used at a computing facility providing batch processing and providing these programs in its internal program library the successive steps in the analysis can be accomplished by submitting a call for a specific program along with a specification of the program parameters and a data deck or tape. Each stage may be submitted separately and the statistician will usually adjust later steps in the analysis from the outcome of earlier computations. The same program, or several different programs, may be used. It may be possible also to store the data in the computer and call it from the file whenever the next analysis is desired. The return of each portion of the analysis may be a matter of hours, perhaps in some cases, a matter of minutes. The data analyst may designate different dependent variables or modify the choice or priorities of the dependent variables, transform the variables, include or exclude cases, or more generally he may choose the definitions of strata.

This analytical search is in some ways analogous to the stepwise regression itself. The art of data analysis has not yet matured to the point where the investigator can specify in advance the algorithms which might be brought to play in this analytical process much less to specify their sequence.

The development of interactive computer systems will greatly enhance the researcher's capacity to ana-

lyze his data. In such systems summary information can be supplied to the user at a console and he can supply commands to the computer from his console. The investigator may therefore intervene in the strategy of analysis, indicating that a run with the same program should be repeated with a specification of new or modified parameters, etc. Thus, this type of system reduces the turnaround time for successive stages of the analysis. Some investigators have found earlier interactive systems to be too restrictive in their available modes of input and output and would prefer a capability additional to that of the typewriter console to provide card input and fast printer output. However, it should be noted that output which requires a great amount of time to read may as well be provided off-line. Also, a very large number of cards may be more easily handled at a central computer.

Computing systems to handle interactive consoles, as well as providing adequate capacity for executing large statistical programs, will soon be generally available. So far the statistician has been required to choose between interactive capacity and capacity to do a comprehensive analysis. He, of course, requires both capacities concurrently. He also needs aid in visualizing the interrelationships of variables and the goodness-of-fit of various statistical models.

Statistical graphs

Many experts in data analysis have always used graphical methods to aid their analysis of data. One often hears directives of these experts to their assistants something like, "go thou and *plot* your data." The plots and charts frequently do not survive the process of report writing and publication, but have played an important part in the analytical process itself.

These charts have employed various colors or symbols to identify strata and lines have been drawn, dotted, dashed and perhaps wiggly. Question marks annotate extreme values. Regression lines or frequency contours may be drawn in alternate forms with and without suspect cases or with and without doubtful subgroups. Various forms of graph paper are used and curves are plotted with standard error regions.

The data analyst who has taken the computer unto himself may have dropped some of these time consuming graphical procedures in favor of various computer aided analyses of alternate approaches. He still, however, seeks graphical output from the programs he uses and still very likely employs some graphical methods in plotting his next analytical move.

Recently the interactive computing systems have been introducing television type screens to supply more rapid output at the user's console. These scopes may also be equipped with a light pen to be used for communicating signals from the user to the computer. These signals may be used as commands to steer the course of the analysis by supplying parameters ,changes in parameters, choice of subroutines and identification of data points, etc. The light pen is slightly larger than a fountain pen and affects the designation of information on the screen by sensing the screen regeneration at a particular position.

The users console may have available also a typewriter, instruction keys and perhaps card input, or more luxuriously such additional input media as paper tape, magnetic tape, disc packs and film. When the console is connected with a computing system that has a well developed operating system, program library and file service, the user may build effective programs by virtue of having his commands executed within a reasonable length of time, having available many packaged service and analytical programs as he works at the console. A few computing facilities are approaching this capacity at the present time. These facilities provide much of what is needed for effective data analysis.

A major difference will be the need to operate at a computer language level at least one step higher than that required for the programmer. The statistician will not in general be able to make sufficiently rapid progress with his analysis if he must do any significant amount of programming while he is "on-line."

Simple scopes which provide only character displays can satisfy needs of the programmer but will not provide the graphical needs of the statistician. Scopes are available which provide plotting capabilities and vector or line drawing features. When a scope of this type is coupled with a light pen to provide a simple form of identification and communication concerning the results observed on the scope and a function keyboard to initiate commands to subroutines and to supply parameter values, etc., the hardware capacity is available for the statistician to construct graphs and to interact with an analysis in progress in the computer in a "plotting" or graphical mode.

The next step in providing an operative system for him is to design the supporting software. The overall system will need to be serviced at several levels of programming. The following paragraphs indicate some of the statistician's needs.

Data storage

A basic data file will be assumed to be structured to an extent that one may arrange the data into a *data matrix* whose columns represent variables (i.e., types of measurements) and whose rows represent individuals or cases on whom the measurements have been made.

It is not assumed that every measurement is available for every case, but that various types of "missingness" are either coded in the original data or provided by a computer program.

Operations on the data file

One or more variables may be selected by name or number. Operations available on this variable (or on several variables) include functional transformation by a function supplied at the console. These new variables may be added to the data matrix or, if necessary, replace other variables in the data matrix to conserve storage space. Stratification of cases can be accomplished by forming submatrices based on categorizing statements involving the data entries for one or more variables. The above features provide, for example, the capacity to compute residuals from a regression function based on the same or other observations. Submatrices or extensions to matrices may be stored as derived files.

It is frequently desirable to perform a transformation on one or more variables where the functional form is known but for which not all parameters are either known or computable from the given data. One may wish to examine the effects of the transformation in a regression or discrimination problem and note the effect on the prime analysis of various selections of the parameters of the transformation. This calls for a convenient way of providing a sequence of values to one or more parameters and to watch at the scope the effect on the plotted outcome of the computations. In some cases it may be sufficient to display conversion indicators in numerical form and guide the process accordingly.

Parameter pacing

A simple replacement of the parameter values one by one at the console is frequently not sufficient to the task. A pacing subroutine needs a starting value and an increment to provide a succession of inputs so that the progression of events may appear as rapidly at the scope as they may be provided by the main computer. This subroutine also provides an "increase" and "decrease" instruction to accelerate or decelerate pacing and a backspace instruction to avoid restarting the parameter search when an optimum point has been passed.

Scope plots

The statistician uses plotting on paper in many ingenious ways to increase his understanding of various characteristics of the data. A change of scale, e.g., from arithmetic paper to logarithmic paper or the use of probit or logit paper for cumulative proportions can be accomplished on the computer by transformations on the data.

The ability to plot data pairs (x, y) on paper including the use of dots or other characters to represent strata is directly transferable to the scope. Although the use of colors for strata is not generally available on today's computers, the use of motion provides an even richer visual presentation. If a particular stratification merely represents an ordered categorization based on a third variable the identification of strata may be viewed as the capacity to introduce a third dimension to the two cartesian coordinate variables for the plane face of the scope.

Let us examine for a moment what can be displayed on the scope. The simplest representation of one variable can be dots along a line or, in two dimensions, a point in the plane. A second variable can be introduced in the plane by locating a line segment at x-horizontally and using a line segment for y (sometimes called a histogram). A third dimension may be added to a point (x, y) by changing the dot size or displaying a line segment whose center is (x, y). The value of a third variable can be indicated by the length of the line. A third variable may also be introduced by using a short line segment of fixed length whose center is positioned at (x, y) but whose angle of inclination represents a third variable.

Stereo representations can be provided which give true stereo with the use of viewing lenses and the impression of stereo by configurations changing shape as though rotating in space. When regeneration is spaced in time and applied successively to subgroups of the points (or symbols, characters, or line segments) attention is drawn successively to each subgroup in turn. The subgroup designation may be a third variable. Subgroups may be differentially highlighted by the frequency or duration of regeneration. If sufficient system support can be given to the console scope, line segments may be given either a metronome motion or rotation whose frequency or speed represents an additional variable. Lengths of line segments may be automatically scaled down from maximum length to give a length proportional to the third variable. The slope of a line segment may represent a residual scaled by the maximum residual or represent a third variable scaled to cover its range between $+1$ and -1. These facilities can be supplied with relative ease to the graphical system. The pictorial display may be alternated with or overlayed by the usual numerical output of the statistical program. The "printed" and pictorial presentation can also be located at different places on the scope.

Background grid

A grid of the usual graph paper type can be easily provided but since color is not usually available, the

grid lines should be of lesser intensity than the data points. A more effective mode for the user is provision of regeneration to the background only at specified times or on call. The user must be able to specify line widths and different frequencies of regeneration. For finer work, provision for second or fifth lines preferably at a different intensity or width should be provided.

Before giving further details on the specification of other modes for displaying additional dimensions, we give examples of the types of variables which one may wish to display.

Since each observation of multivariate data on continuous variables is often conceptually visualized as representing a point in higher dimensional space, the capacity to represent more dimension in the two-dimensional frame is desirable. Categorical data which is ordered can be treated similarly. Special attention may be needed for unordered categories. Preliminary analyses may provide additional dimension to the plotted data. If the plotted points are means, the additional dimension of sample size and standard deviation are important. If percentages are computed on categorical data, the sample size and proportion non-responding are important, etc.

Case identification

The *light* pen may be used to identify a point representing a case. This can cause the computation to be repeated removing this case, showing the resulting changes in the display. A simple example is the display of two alternate regression lines computed with and without one or more designated points (cases). Upon identification of a point and specification of a particular variable, the user can request a histogram showing the entire distribution of that variable with a flashing or highlighted location of the identified case in the distribution.

Program control by light pen

Sequencing of the analysis can be made by light pen or by function keys. Experience has shown that the light pen is easier to use because of the appearance of memory cues and choice alterations on the scope. The pacing of parameters can also be guided by light pen from a choice of the pacing parameters themselves.

Selection of strata

The specification of various strata for analysis can be readily accomplished by calling a selection subroutine whose descriptors are prepared for the particular study. Consider for example a tumor registry which has been prepared using various codes for age, sex, diagnosis, treatment, etc. This subroutine can exhibit first these prime variables. The touch of the light pen to the word "age" appearing on the screen will cause the specific code for age to appear, perhaps showing that age is coded to 5 year age groups. The light pen can be used to touch the desired age groups and return to the original list and proceed. In some cases (depending on which variable is chosen) a choice tree of several levels may be specified by a succession of choices determined by light pen. The computer program constructs a Boolean selection statement which can be used to operate on the data file.

It is fairly obvious that the various forms of case or strata selection can be combined with the various forms of variable selection, the various forms of graphic presentation and the various statistical models of analysis to provide a very powerful tool.

The literature holds fairly complete documentation of the analytical methods available. Systems of data analysis programs are described by W. J. Dixon as Chapter 3 in *Computers in Biomedical Research* edited by Ralph W. Stacy and Bruce Waxman, Academic Press, New York, 1965.

Examples of packaged graphics programs developed thus far using some of the above features are:

1) Simple plot and regression. This program provides data matrix operations including selection and transformation, provides scatter plots with case removal or addition with adjusted regression.

2) Stepwise regression with control of selection and exclusion of variables, transformation, etc.

3) Spectral analysis with series selection, transformation, filter construction and providing spectrum, cospectrum, phase relations, etc.

4) Non-linear regression with specification of function parameters and boundary conditions and control of iterations.

5) File search with code assisted Boolean specification for constructing subfiles with control of descriptions of subgroups.

MEDATA–a new concept in medical records management*

by CAROLINE HORTON, TATE M. MINCKLER, and LEE D. CADY JR.

INTRODUCTION

The potential applications for computers in expediting medical research and improving patient care are well recognized. However, in a medical environment full exploitation of the latent powers of available electronic devices depends on one vital factor. That is, giving the physicians, research scientists, or administrators direct control of the type of information acquired and stored, how it is related, and the timing of acquisition and retrieval. From the viewpoint of these professional users, the paramount goal is to achieve such direct control without sacrificing the valuable time and effort required to become experts in the esoteric art of computer programming and systems operation. This concept of immediate control was the basis for developing MEDATA, an amalgamation of techniques which allows the user to organize, collect, store, and retrieve all types of medical data without resorting to the intricacies of formal computer science.

Organization of data

The MEDATA system does not specify the information structure but provides a framework for processing any structure required for the purposes of the user. Data are collected routinely on some form or questionnaire. Compilations such as abstracts in a library may not use printed forms, but the questions are implied by words such as Title, Author, Journal, and Abstract associated with the data.

Data on forms are usually related by headings and subheadings into a format resembling an outline. In the MEDATA system selected typewriter symbols are used as prefixes to the questions or headings which express and preserve the relationships established by the user. A heading usually represents a broad classification of the questions following it, as illustrated in Figure 1. Occasionally the heading will be additional information to be retrieved with each question, as in Figure 2.

```
$CLIN PATH
    #CBC
        @Hb:
        @WBC:
        @MORPH
            *RBC:
            *WBC:
            *PLATELETS:
    #UA
        @pH:
        @Sp Gr:
        @Alb:
        @MICRO
            *RBC:
            *WBC:
            *BACT:
$X-RAY
```

Figure 1 – Sample program tape

When forms are used, the answers can vary from a numeric quantity to English prose and are frequently a combination of these. The technique of enclosing quantitative or coded data in parentheses at the beginning of the answer permits expression of any type of answer. This gives the physician opportunity for unrestricted expression and the statistician opportunity to retrieve quantitative data.[12]

Methods of data input

A programmed typewriter with an auxiliary paper tape is the input device. This procedure has several advantages. Unlimited numbers of characters may be used for entering data. The transcriber is the secre-

*This project was supported in part by the National Aeronautics and Space Administration Contract No. NSR 44-012-039 from the Manned Spacecraft Center.

The University of Texas M.D. Anderson Hospital and Tumor Institute, Texas Medical Center,
Houston, Texas.

```
W H A L E R S  H O S P I T A L
New Bedford, Massachusetts
MEDICAL SUMMARY
$ID NO:
$NAME:
$DATE:
$PURPOSE OF EXAM:
    #PRESENT ILLNESS:
    #DIAGNOSIS:
    #TREATMENT:
    #COMMENTS:
$EXAMINING FACILITY:
$PHYSICIAN:
```

Figure 2 – Program tape for medical summary

tary, who is accustomed to using the information and is familiar with the notations and spelling. Coding sheets, which are laborious to prepare and proofread, are eliminated. The typed copy produced simultaneously with the data tape satisfies the legal requirements for preservation of a document signed by the physician.

The programmed typewriter is used to define the hierarchical structure for the questions as well as to enter the data collected. To enter the format of a new data collection form into the system, the questions in their outline form are typed and simultaneously punched into a paper tape. This paper tape uniquely defines the collection form and is used to control the typewriter and paper tape punch each time data are collected on this questionnaire. The position of the question in the outline format is flagged by preceding it with one of the selected typewriter symbols. Questions may be grouped in subdivisions to nine levels. The typewriter characters selected to relate these subdivisions are referred to as position codes. These codes are $, #, @, *, ?, ¢, ±, ', %. A colon (:) and typewriter programming codes are entered after each question requiring data. Figure 1 is an example of part of a laboratory collection form showing several levels of subdivisions.

Any headings or typewriter formatting characters, such as the name of the form or institution, can be entered preceding the first position code and will affect the appearance of the typed document but will not affect the storage and retrieval program. When the specific tape created for the form is used as a control tape, special headings are copied onto the paper tape being punched. Figure 2 is an example of a program tape used for medical summaries showing the outline headings and the position codes.

The position codes, the question, and the colon are copied from the control tape to the new paper tape. The question and colon are typed on the document. If data are to be entered, the typewriter programming codes after the question will cause control to be transferred to the secretary. Data entered through the keyboard will appear on the document and will be punched into the new paper tape. Quantitative or coded data are enclosed in parentheses at the beginning of the entry. Any parentheses after the first character of the data are assumed to be part of the prose data. Figures 3 and 4 are medical summaries collected using this procedure.

W H A L E R S H O S P I T A L
New Bedford, Massachusetts
MEDICAL SUMMARY

ID NO:	12345
NAME:	Mobius, Richard
DATE:	14Dec66
PURPOSE OF EXAM:	Medical Complaint--Limp
PRESENT ILLNESS:	Several years ago noticed severe limp after ocean voyage. PE shows absense of left lower extremity.
DIAGNOSIS:	(Y902-4991-0000)* Traumatic amputation (whale bite) left lower leg, old.
TREATMENT:	Tender loving care
COMMENTS:	Continue follow-up
EXAMINING FACILITY:	Whalers Hospital, New Bedford, Mass.
PHYSICIAN:	David Jones, M.D.

Underlined words were typed automatically by programmed typewriter under the control of the program tape described in Figure 11.

*SNOP coding:[3]

Topography	Y920	Lower left extremity
Morphology	1472	Traumatic amputation complete
Etiology	4991	Whale
Function	0000	Not applicable

Figure 3 – Medical summary – 14 Dec 66

This subsystem for data collection has been formally titled STAT for systematized terminal acquisition technique and is being prepared in detail for publication.[13]

Storage of data

The method of storing data on the mass storage medium departs radically from current data handling techniques. Both the question and its answer are stored for each set of data. This is an extravagant use

```
                    W H A L E R S   H O S P I T A L

                        New Bedford, Massachusetts

                              MEDICAL SUMMARY

ID NO:                      12345

NAME:                       Mobius, Richard

DATE:                       01Jan67

PURPOSE OF EXAM:            Medical Complaint "not feeling well,
                            Doc"

     PRESENT ILLNESS:       About 3 hours following consumption
                            of one-half barrel spiced Jamaican
                            rum, patient noticed onset of blurred
                            vision, occasional dizziness, nausea,
                            and acute heartburn.  Exam showed
                            elderly white male in minor distress
                            without significant physical findings.

     DIAGNOSIS:             (0000-0000-0000-7236)* Acute overdose
                            of spices

     TREATMENT:             Gelucil PRN
                            Advised patient to use unspiced rum
                            in future.

     COMMENTS:              Return for follow-up one week.

EXAMINING FACILITY:         Whalers Hospital, New Bedford, Mass.

PHYSICIAN:                  David Jones, M.D.

Underlined words were typed automatically by programmed type-
writer under the control of the program tape described in
Figure 11.

*SNOP coding:[3]    Topography  0000  Not applicable
                    Morphology  0000  Not applicable
                    Etiology    0000  Not assigned (Spices)
                    Function    7236  Food Intolerance
```

Figure 4 – Medical summary – 01Jan 67

of storage facilities, but it allows extreme flexibility, which is the primary objective. If the question precedes each answer, the information may be retrieved by restating the question and doing a phrase search of the mass memory. Any question can be added to the system by inserting it on the tape controlling the typewriter. No changes are required in the storage or retrieval programs. The physician and his secretary have complete control. Facsimile storage, or FACS, as this data management concept is called, will be discussed in detail elsewhere.[13]

Retrieval of data

Extraction of pertinent data from the file is as flexible as the storage of the data. The terminal typewriter or card reader is used for medical requests based on four questions: WHO?, WHAT?, WHEN?, I/O?.

WHO? may be the patient's name or ID number for medical records or the author's name for library purposes. If more than one name is required, they can be entered in a series separated by semicolons. If selected data are to be retrieved for every name in the file, the request is answered by the word "ALL."

WHAT? may have answers ranging from one item to a complete medical record. If a complete record is required, the name of that record is entered. A single item or group of items can be retrieved by entering the name of the record, a hyphen, and the name of the item

```
WHO     12345.  MOBIUS, RICHARD.
WHAT    SUM-PURPOSE OF EXAM.
WHEN    ALL.
I/O     TYPE.

12345          MOBIUS, RICHARD          14DEC66
MEDICAL SUMMARY
   PURPOSE OF EXAM:     MEDICAL COMPLAINT--LIMP
      PRESENT ILLNESS:  SEVERAL YEARS AGO NOTICED SEVERE LIMP
                        AFTER OCEAN VOYAGE.  PE SHOWS ABSENCE
                        OF LEFT LOWER EXTREMITY.
      DIAGNOSIS:        (Y902-1472-4991-0000) TRAUMATIC AMPUTATION
                        (WHALE BITE) LEFT LOWER LEG, OLD.
      TREATMENT:        TENDER LOVING CARE
      COMMENTS:         CONTINUE FOLLOW-UP



12345          MOBIUS, RICHARD          01JAN67
MEDICAL SUMMARY
   PURPOSE OF EXAM:     MEDICAL COMPLAINT "NOT FEELING WELL,
                        DOC"
      PRESENT ILLNESS:  ABOUT 3 HOURS FOLLOWING CONSUMPTION
                        OF ONE-HALF BARREL SPICED JAMAICAN
                        RUM, PATIENT NOTICED ONSET OF BLURRED
                        VISION, OCCASIONAL DIZZINESS, NAUSEA,
                        AND ACUTE HEARTBURN.  EXAM SHOWED
                        ELDERLY WHITE MALE IN MINOR DISTRESS
                        WITHOUT SIGNIFICANT PHYSICAL FINDINGS.
      DIAGNOSIS:        (0000-0000-0000-7236) ACUTE OVERDOSE
                        OF SPICES
      TREATMENT:        GELUCIL PRN
                        ADVISED PATIENT TO USE UNSPICED RUM
                        IN FUTURE.
      COMMENTS:         RETURN FOR FOLLOW-UP ONE WEEK.
```

Figure 5 – Purpose of exam retrieved

or group to be retrieved. Figure 5 illustrates the information retrieved by a request for all summary information on a patient. SUM is the name of the record of MEDICAL SUMMARY data. If a common item is to be retrieved from several records with different names, such as the diagnoses from all hospital records for the patient, the question is answered ALL-DIAGNOSIS. Figure 7 shows how ae item of information can be retrieved from charts originating in several unrelated offices with no limitations imposed on any of the offices. If selected records are to be retrieved depending on content, the question is answered by the name of the record, hyphen (-), the question, colon (:), and the required series of symbols. Figure 6 shows the selection of the chart with a specified phrase in the diagnosis.

```
WHO     ALL.
WHAT    SUM-DIAGNOSIS:  WHALE BITE.
WHEN    01DEC66 - 28FEB67.
I/O     TYPE.

12345          MOBIUS, RICHARD          14DEC66
MEDICAL SUMMARY
   PURPOSE OF EXAM:  MEDICAL COMPLAINT--LIMP
   DIAGNOSIS:        (Y902-1472-4991-0000) TRAUMATIC AMPUTATION
                     (WHALE BITE) LEFT LOWER LEG, OLD.
```

Figure 6 – Diagnosis retrieved

The user-designed system of headings and subheadings has a particular advantage in handling medical data. For example, if information on *eyes* is requested, the data are returned with all superior and inferior headings – *pupils* belonging to *eyes*, under

```
WHO      MOBIUS, RICHARD.
WHAT     ALL-DIAGNOSIS.
WHEN     01JAN67 - 28FEB67.
I/O      TYPE.

12345          MOBIUS, RICHARD          03FEB67
DENTAL REPORT
   DIAGNOSIS:      GUMS AND REMAINING TOOTH HEALTHY.

12345          MOBIUS, RICHARD          28FEB67
EYE EXAMINATION
   DIAGNOSIS:      POOR EYES, MAY CHASE WHALES WITH GLASSES ONLY.

12345          MOBIUS, RICHARD          01JAN67
MEDICAL SUMMARY
   DIAGNOSIS:      (0000-0000-0000-7236) ACUTE OVERDOSE
                   OF SPICES.
```

Figure 7—Diagnosis retrieved

the major title, *head.* This is necessary for questions like WBC (white blood cells or count) which may be found in blood, serum, urine, etc. Data on WBC are retrieved with all the headings under which it appears. Figure 8 shows the retrieval of an item and its identifiers from several sections of the chart shown in Figure 1. LAB is the record name for the LABORATORY SCREENING TESTS.

```
WHO      MOBIUS, RICHARD
WHAT     LAB-WBC.
WHEN     06MAR67.
I/O      TYPE.

12345          MOBIUS, RICHARD          06MAR67
LABORATORY SCREENING TESTS
   CLINICAL PATH
      CBC
         WBC:  10,200
         MORPH
            WBC:  NORMAL
      UA
         MICRO
            WBC:  NEGATIVE
```

Figure 8—WBC retrieved

WHEN? may be a specific date, a series or range of dates, or all dates. Figures 5, 6, 7, 8 illustrate this.

I/O is the choice of the available output units: TYPEWRITER, PRINTER, MAGNETIC TAPE, or DISK depending on the application and availability.

This data storage and retrieval system can be used as a FORTRAN subroutine in developing the data tapes for more elaborate tabular or mathematical statistical manipulations which users may wish to undertake with existing packaged statistical programs. Of course, special programming is required, but the task is more than half finished when the data are available in computer-readable form. A complete description of this natural inquiry language for retrieval is in preparation.[13]

Computer requirements

Computer requirements for the system are minimal. The system is written in an extremely basic FORTRAN using only READ, WRITE, DO statements, GO TO statements, computed GO TO statements, arithmetic IF statements, arithmetic statements, and COMMON. All communications between subroutines are done through COMMON to achieve maximum speed and minimum core requirements. As real-time multiterminal monitors become available, the FORTRAN program can be modified to accept requests from terminals.

MEDATA was developed using an SDS 930 with a console typewriter and magnetic tapes for storage. An IBM 1050 is the programmed typewriter used in this system.

Plans for use of MEDATA concepts

The system is being used for processing medical data on the astronauts at NASA Manned Spacecraft Center, Houston, Texas. The basic techniques are applicable for literature storage and retrieval, administrative records, and many other automated data processing requirements of a medical institution. Currently, the MEDATA methodology is being adopted for storage and retrieval of all personnel data for the staff of The University of Texas M.D. Anderson Hospital and Tumor Institute. Preliminary demonstrations have been conducted as the first step toward applying MEDATA techniques for all medical records in the institution.

The system has received enthusiastic support from the professional staff using the techniques and those who have participated in demonstrations. The operations are quickly assimilated by persons with no background in computer science, and the users appreciate a sense of full control.

Secretarial or clerical personnel were easily and rapidly trained to create basic master tapes from the medical records and insert initial data as well as add subsequent observations. The only prerequisites are familiarity with mechanical operation of the programmed typewriter and reasonable typing skill. Typographic errors are quickly corrected by deleting incorrect entries and typing the correct information. Querying the computer using the WHO, WHAT, WHEN, I/O technique can be accomplished by anyone—typist or not—after five minutes of training. Neither the physician nor the typist needs to learn a new language to use the computer.

Discussion

The pattern of progress in applying computer sciences to everyday problems may be likened to the

pattern of growth in an Airdale puppy. Each extremity seems to develop independently, and not until maturity is reached does the beast present an appearance of functional integrity. The advent of time-sharing technology is focusing increasing attention on the most retarded extremity of computer science—providing the unsophisticated user access to computer power. For the concept of time-sharing to realize its full potential, the computer must become a functioning part of the user's environment. The capabilities of computer connected terminals are multifold and include not only the control of central computer analyses and communication facilities but also provide significant improvements in data collection and retrieval options. The developments reported here explore the integration of new ideas and available equipment to produce a user-oriented medical data collection, storage, and retrieval system.

During the conception and embryogenesis of MEDATA, a concerted effort was made to utilize the applicable published ideas in the field. The integration of a terminal into the system for acquisition and retrieval of data by the medical user has been employed by others.[2,9,14] The advantages of direct input by the secretarial personnel through a programmed typewriter terminal was adopted by the National Library of Medicine MEDLARS activity,[1] and elsewhere.[12] User-oriented data organization and variable field structuring are features of several medically-oriented systems,[4,6,7,8,10] and the user-oriented retrieval language (Who, What, When) was first published by our own group a year ago.[5] The amalgamation of these approaches to the various facets of medical records automation resulted in MEDATA, which offers new dimensions to the field.

SUMMARY

MEDATA, an automated medical records system, represents an organized approach to the collection, storage, and retrieval of medical data. This completely user-oriented system takes advantage of the background and training of the medical secretary and simultaneously by-passes the conventional keypunch operators and coders. It is capable of responding directly (in his own terminology) to the user who lacks extensive computer background. The system is inexpensive to maintain, and the programs and basic system concepts are machine independent.

REFERENCES

1 C J AUSTIN
Data processing aspects of MEDLARS
Bull Med Libr Assn 52(1):159-163 1964

2 G O BARNETT P A CASTLEMAN
A time-sharing computer system for patient-care activities
Computers and Biomedical Research 1(1):41-50, March 1967.

3 College of American Pathologists Committee on Nomenclature and Classification of Disease
Systematized nomenclature of pathology (SNOP)
Chicago Illinois

4 R H DREYFUS W J NETTLETON JR, J W SWEENEY
Tulene information processing system—version II
Tulane Univ Comp Sci Series Monograph Number 3 1966

5 W F FARLEY A H PULIDO T M MINCKLER L D CADY JR
Man-machine communications in the biological-medical research environment
Proc 21st Nat ACM Conf Assoc for Computing Machinery pp 263-267 1966

6 C HALI C MELLNER T DANIELSSON
A data processing system for medical information
Meth Infor Med 6(1); 1-6, 1967

7 J KOREIN A L GOODGOLD C T RANDT
Computer processing of medical data by variable-field-length format—II
Progress and Utilization of the Technique. N. Y. Univ. Med. Cent. pp. 1-16, June 1965.

8 B G LAMSON B C GLINSKI G S HAWTHORNE J C SOUTTER W S RUSSELL
Storage and retrieval of uncoded tissue pathology diagnoses in the original english free text form, appendix A
In user's Manual I—A Natural Language Information Retrieval System. IBM Data Processing Division of Los Angeles Scientific Center, November 1966

9 L J LAULER W D FULLER
The Lockheed Hospital information system
Fifth Annual Symposium on Biomathematics and Computer Science in the Life Sciences, Houston, Texas (abstract). pp 70-74, 1967

10 L B LUSTED
Primate record information management experiment (P.R.I.M.E.)
Oregon Regional Primate Research Center 1966

11 T M MINCKLER R K AUSMAN T GRAHAM G R NEWELL P H LEVINE
HUMARIS—An automated medical data management system
Meth Info Med 6(2):65-69, April 1967

12 T M MINCKLER
The ANTICS of medical coding
Presented at annual meeting ASCP-CAP 27 Sept 67 Chicago, Ill.

13 T M MINCKLER C L HORTON
Automation of medical records—the MEDATA system
Chapter in NASA Publication Uses of Computers in the Life Sciences (in prep)

14 M V SLACK B M PECKHAM L J VANCURA W F CARR
A computer based physical examination system
J.A.M.A. 200(4):224-228, 1967

Requirements for a data processing system for hospital laboratories

by IRWIN R. ETTER
The Mason Clinic
Seattle, Washington

INTRODUCTION

The modern clinical laboratory should employ automation wherever it is applicable in a continuing effort to meet the demand for ever-increasing amounts of analytical information. The laboratory must also keep pace with the latest fruits of medical research which must be added to the clinical diagnostic armamentarium. This laboratory information explosion imposes great demands upon the laboratory staff. Little hope exists for expanding the staff to meet all these demands. Instead, more efficient use must be made of trained personnel. This can best be obtained by automation of clinical test procedures and automation of the accompanying paper work.

Much has been accomplished in the automation of test procedures. Several sets of equipment are now commercially available which relieve the laboratory technician of the tedium of handling each sample at all stages of analysis. Many more will be forthcoming in the near future as instrumentation manufacturers realize the great potential market size and as new instruments become available.

While new automatic instruments perform chemical analyses and relieve the bottleneck at one stage of the laboratory information flow system, if not joined with computers they tend to impose another bottleneck immediately downstream. This is due to the large amounts of time required to perform calculations on the resulting laboratory data and to perform the clerical procedures required by the hospital. At this one facility it has been estimated that the ratio of time spent on paperwork versus true technical work approaches 3:1. The greater the degree of an instrumental automation the larger the ratio.

Typically, the hospital laboratory staff must translate voltage values obtained on an instrument into meaningful values for the clinician, expressed in concentration per unit volume. This alone requires tedious mathematical or graphical manipulation in which gross errors occur due to fatigue. In parallel with the calculation stage, the staff must prepare daily log sheets, laboratory summary and statistical reports, quality control reports and billing information. All of these procedures are clearly within the capability of currently available computing systems. The major difficulty is one of obtaining operating systems at costs that will be economically justifiable. As computer prices drop such systems should become available.

The overall general requirements may be summarized as follows:

1. To minimize routine data processing by laboratory personnel.
2. To reduce error inherent in high volume routine.
3. To provide better utilization of trained technologists for research and development.
4. To reduce data processing time.
5. To provide documented quality control data.
6. To provide predetermined and invariable parameters for acceptable quality of standards, instrumentation drift and deviation of pooled samples, documented during test runs.
7. To provide daily, weekly, and patient summary reports for laboratory, accounting, and clinician uses.
8. To provide these services on-line during the time of actual specimen processing.
9. All of this fully justified economically.

Requirements imposed by instrumentation

Most hospital and/or clinic laboratories have three types of samples to run: routine (hospital), emergency (hospital and clinic) and routine (clinic). Routine hospital samples must be reported out of the laboratory within one day after having been drawn (some specialized tests can take longer). Emergency samples must be reported as soon as possible. This is especially true of

hospital samples where test results are a major factor in determining patient treatment. In a clinic, fast response may be required for some diagnoses, but a one-day interval is often sufficient. Ideally, in all cases, only a few hours elapsed time between sampling and reported results should exist. This requires that any data processing system used to process laboratory results should be on-line with the instruments and should be capable of producing results as soon as the sample is run. A laboratory data processing system must interface with two varieties of instruments: automated and manual. Automated instruments operate at a variety of speeds from high frequencies of 1000 cycles/sec. to slow instruments of 1/10 cycle/minute. The speed of manual instruments depends on the rate at which samples can be placed in the instrument and the length of time required for the instrument to reach a steady state reading for the sample. Most manual instruments produce data points at a frequency of about 1/minute. A variety of approaches can be used to monitor the operation of the instrument.

Two basic approaches can be used for the manual instrument. Since an operator is required to be at the instrument while it is running, he can either record the readings and then enter them into the data processing system, or he can signal the system to read the signal on the instrument automatically. Each of these approaches has some advantages. The first approach of recording the data and entering it together with patient number into the system after a run is finished is the least expensive of the two methods considered. A standard keyboard (e.g., teletype or Selectric) typewriter can be used. Since all of the information is entered in digital form, signal processing is not required. Human transference of numbers allows room for error and is the major drawback in the approach. This approach will also be more time-consuming. The more sophisticated approach is to have the computer system joined to the instrument. When a sample is placed in the instrument, it can generate a signal which causes the system to read the instrument signal and search successive readings for a steady state level. This approach is free of error from human data handling and entails less operator time, but may be twice as expensive as the first approach. Further pilot evaluations of the relative economics, accuracy and speed will be the deciding factor between the two approaches.

The critical factors in evaluating the approaches to be taken to handle the automated instrument are frequency content and duration of the signal. Most instruments fall into three groups: High frequency-short duration, medium frequency-medium duration and low frequency-long duration.

High frequency-short duration instruments are typified by high speed mass spectrometers and cell counters. At very high speeds, many data points are generated. Often the scan is repetitive and can be displayed on an oscilloscope. The best way to handle these instruments is to store rapidly and continually the raw data by sampling the signal and converting it at high speed and to process the data at some later time. This is effectively slower than real-time computer operation. An alternative approach is to record the data on high speed magnetic tape for slow replay into the computer. This is probably a less expensive approach since high speed input devices for the computer system can be eliminated. Fidelity of the signal is reduced very slightly by using the tape intermediate, and the costs are certainly reduced. Permanent analog record of the data is also provided in this approach. For some laboratories and instruments this may be an important feature.

Medium frequency-medium duration instruments are typified by the "Auto-analyzer" and gas chromatograph. For instruments in this range, real-time data processing is possible. In this mode, the instruments are monitored about once each second, the raw data being processed as it becomes available. In some situations, short delays in processing may be necessary when exceedingly long routines are required. The effect seen by the laboratory staff has been one of real-time operation.

On-line data processing is highly inefficient for low frequency-long duration instruments typified by amino acid analyzers for which a run may be several hours long, with significant data being developed about once each hour. For these instruments tape recording of data with fast replay into the computer seems to be the best approach. Signal fidelity will be maintained because the signals are of low frequency. Since very few tests run on these long-running instruments are urgently required by the medical staff, the computer can be scheduled to process these tapes during its idle periods (e.g., evenings).

The varied requirements imposed by the laboratory instruments are:

1. Manual entry station for off-line manual instruments.
2. High speed data input channel for on-line high speed instruments.
3. Low speed input channels for medium speed instruments.
4. Analog tape input for fast replay of slow instrument signals and slow replay of fast instrument signals.

Requirements imposed by hospital personnel

Information stored in the data processing system must be readily available to hospital personnel. Laboratory staff, doctors, and research workers all will require the capability of entering information into the system and of interrogating the system as to its contents. These data transfers should require little or no typing skill on the part of the personnel and should yield reports with a minimum of excess information.

The laboratory staff must enter information about test requests into the system. In addition, queries as to the status of on-line instruments and the contents of patient files must be entered. Similarly, information about the test procedures will require modification as procedures are changed in the normal course of laboratory development. Routine reporting of test results on each patient and documentation of laboratory work performed will be initiated by the system itself or they may be requested by laboratory personnel.

The doctors' prime requirement of the system is a supply of up-to-date information about their patients. Test requests are initiated by the doctors who desire clear, concise laboratory reports of the results obtained and their statistical significance with respect to hospital norms. In addition, the possibility of entering significant clinical observations into the patient record should be provided to allow clinical researchers to have as complete a file on each patient as possible. The laboratory data processing system is small and cannot be used to provide a patient record-keeping service. This task should be considered for the hospital information system which is the central information system.

Medical research personnel will require the data processing system to have the capability of being used as a limited capacity, general purpose computer. In addition, those involved in the development of laboratory techniques will require access to data developed in the laboratory in order to evaluate their ideas.

The requirements imposed by the hospital personnel may be summarized as:

1. Flexible information input capability for both fixed and variable format data.
2. Multiple output units with a wide variety of formats.
3. General purpose, as well as special purpose capability.
4. Rapid data retrieval.

Hardware configuration

The description of a system that will satisfy the foregoing requirements is intended as a guide to the perplexed, both the manufacturers and the users of data acquisition systems. Several systems are capable of performing the data acquisition task. Factors of cost and reliability dictate the use of small systems composed of a minimum of devices. Our experience is proposed as a guide to the fundamental elements desirable in a hospital laboratory data acquisition system. At the heart of the data acquisition system is a small computer. The computer acts as both system controller and data reduction center. Several small-to-medium sized machines, currently on the market, have sufficient capability to meet the requirements. The major hazard in selecting a computer is extremism. Too small and/or slow a machine will limit the ultimate capabilities to which the system may be expanded. Too large a machine is a waste of capability. Computer specifications should fall within the following capabilities:

Memory Size ..... 4-16K Core
Arithmetic Unit ..... Hardware-Add, Substract-less than 3 μ sec.
Hardware-Multiply-Divide-less than 30 μ sec.
Hardware Index Registers
Real-Time Clock
Flexible Input/Output Structure
Direct Memory Access Channel
Data Channels
Program Controlled Data Transfers
Program Interrupt

It should be stressed that the instruction repertoire need not be very large since the system will mainly contain one fixed program. It should not, however, require a major effort to modify the program. Reliability of the computer is essential to efficient on-line operation. The computer is potentially the most reliable component of the system and it should be expected that computer failure be a rare occurrence.

Peripheral devices (i.e., tape recorders, typewriters, printers) have been of notoriously low reliability. This dictates the use of minimal number of peripherals and wherever possible some duplication. The system should be designed to allow operation to proceed despite any peripheral failure.

In any data acquisition system of more than minimum size, some mass storage device is necessary. Exceedingly high speed access is not of major importance. Reasonably long times (up to 1 sec.) can be tolerated to retrieve a record from the mass storage device. The majority of this time would be spent in finding the location of the record in the mass storage and only a short time spent on actually performing the transfer. The processing system must be capable of sending a request to the mass storage de-

vice and then continuing operation until the actual data transfer takes place. Reliability is of great importance here since system operation would be drastically curtailed if the mass memory unit became inoperable.

The basic communication channel between the system and the hospital personnel is through the keyboard typewriter/printer unit. The most widely used unit is the teletype. At least two are highly recommended and the most rugged versions should always be used. One unit should be equipped with a paper tape punch and reader. Only in very large systems will a high speed line printer be worthwhile.

A slow speed card reader should also be included. Information entered on cards includes: test type information, to be entered only when modified, and any standard input information to relieve the need for typing. It is expected that the card reader will occasionally not be functioning properly.

The analog and digital interface which allows direct monitoring of laboratory instruments is composed of scanners or multiplexers, A to D converters, and a set of digital logic for testing instrument operating conditions. Two analog input systems can be used for fast and slow input respectively. The slow speed unit should be capable of accessing 200 points/sec. while the high speed unit should be capable of at least 10,000 points/sec. Both units should operate under program control. The digital interface should be able to detect voltage above a preset level and switch closures, generating logic levels 1 and 0 for true and false states respectively. In addition, the capability of reading a set of voltages or switch closures in groups equal to computer word size and organizing them into computer words is essential.

The signals on most slow and medium instruments can be taken from the swinger of a retransmitting slide-wire mounted on the recorder. The swinger is connected to the input of the scanner. The scanner in the slow speed input unit is sequentially connected to each position. At each position are several signals: 1) a ground reference, 2) the signal from the swinger, and 3) any logic that may be used to test instrument conditions. From the scanner, the analog signal is sent to the analog-to-digital converter which is under program control.

The digital interface consists of a battery of logic elements which interrupt the computer when instruments need to be serviced. Instrument priorty may be established either by hardware logic (e.g., a daisy chain in which instruments connected closer to the system have higher priority) or by software routines which check a sequence of interrupt sources; the sequence establishing the priority. Also included in the digital interface are control units located at the instrument. These units are used by the laboratory personnel to inform the system of the status of the instruments at that location. In addition, any manual data entry stations are processed by a digital interface which essentially reads a special purpose keyboard.

Software configuration

The key element in obtaining a working data acquisition system is the software. Many shortcomings in the hardware can be offset by creative programming. There are basically four parts of software: executive routines, data processing routines, bookkeeping routines, and communication routines.

The executive routines are responsible for controlling the operation of the system. The order and timing for servicing interrupts are its most important responsibilities. Many different approaches to priorities can be used. One simple approach is to perform a minimal amount of processing on each interrupt as it occurs. The interrupts which are waiting and have not been completely serviced are then checked to determine which is oldest and the oldest is then processed. An alternative is to develop some method of deciding how much is to be done with each interrupt and handle the shorter ones before the longer ones. Another scheme which has merit is to assign levels of processing required of each interrupt, e.g., level 1 will be storage of raw data, level 2 will be data smoothing, level 3 peak picking, etc. Level 1 is performed on all interrupts. Then, if time permits before the next interrupt, any level 2 processing jobs will be done. Certain levels like level 3 will result in assigning higher levels to the data (e.g., performance of standardization or conversion of a peak value to concentration units) or will result in completed processing of that piece of data. It is assumed in all these priority schemes that there is sufficient time to process all the interrupts since there will be slow periods when less work is required by the system. The executive routines also control the sequencing of the printout routines, assigning highest priority to those messages which require some action on the part of the laboratory staff (e.g., a warning that the standard samples are not in the proper ranges) and lowest priorty to printing long summary reports.

The data processing routines are called in a variety of sequences by the executive routine to perform all mathematical manipulations of the data stored in the computer. For most on-line instruments, the data processing steps of "smoothing" and "peak-picking" are common to several types of instruments, while

others such as the calibration and unknown determination routines may be specific to one type of instrument. Development of these data processing routines requires a thorough understanding of the laboratory methods. Also, included in the data processing routines are a group of subroutines that are used by many different routines. Such subroutines as floating point addition, multiplication and division, binary to Binary Coded Decimal (BCD) and BCD to binary conversion routines, transcendental functions, and various logic routines that are not performed by hardware registers are included in the set.

Bookkeeping routines perform all the data sorting required by the system. This includes sorting test requests into lists of samples to be run on each test type, relating calculated data on an instrument run to the patient's record where they are to be filed, and assembling data in formats required for typeout routines. No data manipulation is performed by these routines except for data conversion between the coding required by the input/output peripherals and the binary coding of the computer.

Communication routines perform all the operations of acquiring data into the system from the instruments and the peripherals, and developing reports for the laboratory and hospital staffs. These routines include the testing of instrument status (on, off, test type, samples expected, etc.), obtaining analog data from the instruments in the digital form required by the computer, obtaining information entered through the teletypes and card reader, printing of laboratory work reports (instrument load tests, daily work reports, billing reports), and patient record reports on a test request basis and on a summary report basis.

In the normal mode of operation, the executive routine resides in core and calls routines as needed from bulk storage into core. Sections of core are also reserved as transfer stages between core and bulk storage and between core and peripheral devices. In addition, all data of current interest to the system are stored in core including fixed information such as values of test standards and the information currently being developed by the instruments. The executive routines also reside in bulk storage to eliminate the need of loading it through the paper tape reader. Initially, a small paper tape routine is entered into the computer which when executed enters the executive routines into core and starts the operation. The executive routine need not be entered into core again unless it is determined that part of the routine has been lost.

All software routines are permanently stored on paper tape and cards. In addition to the full system's normal routines, several operable routines of a more limited scope are available to avoid using particular components of the computer should any part of the system be out of operation. Thus, if the bulk storage is inoperable, a routine can be used which merely processes the data as it is developed on the instrument and produces reports based on laboratory runs. Data sorting according to patient is eliminated as is summary reporting. All data produced during this period are stored on paper tape so they can later be entered into patient records when the bulk memory is again operable.

System development

Since the system described here is large and the operation complex, it is recommended that development be done stepwise. This will allow the laboratory staff to gain confidence in the system's capability and not frighten them with a large "monster" they cannot understand. Laboratory personnel should be closely involved in the development to aid in obtaining acceptance of the system. The cost of such a system is relatively high for most hospital laboratories and the stepwise acquisition of the system greatly eases the strain on the budget. Almost all that is done during the early stages will be of use at later stages.

The first stage is the development of a unit which will acquire data from several instruments of the medium speed variety in an on-line, real-time mode. No patient identification or sorting is performed by the system at this stage. All results are in the form of a report on an instrument run. (Correlation between position on the run with patient is still done by hand.) The report, developed at this stage, will be used in later stages as a laboratory work report to provide documentation of laboratory procedures. At present pricing levels, cost of this initial system can range between $35,000 and $100,000 depending on the size of the computer and the size and type of interface (instrument and human) developed.

This level is a convenient point to use on the first stage since it will not appreciably disturb the information flow in the laboratory and will relieve the laboratory staff from hand mathematical processing of instrument data. Since little peripheral equipment is involved (teletype, analog, and digital interface) reliability of the system can now be proven. Laboratory staff should become involved with the system to establish complete confidence in the system's capability.

Further development is aimed in two directions, increasing the number of instruments tied into the system and increasing the amount of the paper work load

assumed by the computer. As new instruments are put on-line with the system, thorough testing of the data processing must be performed. Addition of peripherals such as a card reader will enable the computer to assume the job of preparing load lists for the instruments. Bulk storage will allow for the keeping of patient records and reporting of results on a patient-by-patient basis thus eliminating two of the major paper work jobs now performed by laboratory personnel.

The major guidelines at all times should be proven capability, system reliability, economic justification for expansion, and maintenance of system flexibility. At present a reasonably sized data acquisition system for a hospital laboratory will be capable of accepting test requests on punched cards, providing instructions to laboratory personnel for the running of instruments, automatic retrieval of data from about twenty automatic instruments and several manual instruments. It will also provide documentation for the laboratory, the medical staff, and the billing office in terms of test results, individual and summary reports on each patient, and amounts to be charged to patient accounts. While little or no attention to the inner workings of the system should be required during normal operation, in-house capability to modify the computer system will become desirable if the unit is to keep pace with laboratory developments. At present price levels in the computer industry, such a system as described here will cost approximately $250,000. This is more than the market (with a few exceptions) will be able to afford. A price tag closer to $150,000 will be more acceptable for a system that is completely assembled with a complete set of operating software.

As hospital computer systems of several varieties are developed and become accepted pieces of equipment, an integrated hospital system will be required. Each separate unit such as the laboratory data acquisition system should then act as a satellite unit exchanging information with a central hospital information system. The need for a separate laboratory unit will then become especially great. The central hospital system will be occupied with retrieving, assorting, and supplying information to and from all parts of the hospital and should not be required to devote large amounts of time to processing laboratory information. With a laboratory data acquisition system preprocessing the laboratory data into a form that is compatible with a central information system, the laboratory will be able to make highly efficient use of the central system.

An advanced computer system for medical research

by WILLIAM J. SANDERS, G. BREITBARD, D. CUMMINS, R. FLEXER, K. HOLTZ, J. MILLER and G. WIEDERHOLD
ACME Project, Stanford Medical Center
Stanford, California

INTRODUCTION

The ACME project

The Stanford University School of Medicine is located on the main campus of Stanford University, in Palo Alto, California. It was moved from San Francisco to the Palo Alto campus in 1959, with the purpose of more closely integrating medical research and education with the other activities of the University. It shares, with other departments of the University, the computing facilities of the Stanford Computation Center. These facilities include an IBM 7090, a Burroughs B-5500, and a recently delivered IBM/360-67. In addition, there are currently four PDP-8, four LINC, and one LINC-8 computers in use within the medical school. Although this collection of computers represents a great deal of computing power, its distribution was such that the research needs of the medical school were not being fully met.

The Stanford Computation Center is dedicated to serving the broad needs of the University community. With the current batch-processing systems on the 7090 and B-5500 and the planned time-sharing system on the 360-67, the Computation Center is obliged to provide general-purpose computing to a large number of users with quite diverse interests. Inevitably, the needs of any special group cannot be entirely satisfied. In particular, many of the needs of the medical research program are such that a general purpose computing system is not satisfactory. Among these needs are:

1. Analog-to-digital and digital-to-analog conversion
2. Real-time data collection and analysis
3. On-line control of experiments
4. High-speed data acquisition and distribution
5. Support of satellite computers.

The small computers within the medical school provide some relief, in that they are equipped for analog-digital conversion and are being used for data collection and analysis and for control of experiments. However, being small and having few peripheral units, they are limited in the amount of data analysis they can do economically. They are also more difficult to program, since their software is generally quite primitive. In addition, not all research projects can afford the luxury of having their own computer nor the specialized manpower to program it.

In order to determine how the needs of the medical school research program could best be met, a medical school computer policy committee was formed. The committee worked actively with the medical school computer user's group, the Stanford Computer Sciences Department, and the Stanford Computation Center. The result of the committee's work was a proposal for the ACME (Advanced Computer for Medical Research) Project. Initial funding for the Project was a planning grant from the Josiah Macy Foundation, while ongoing support has been provided by a grant from the National Institute of Health, the Macy Foundation, and sharing of costs with other projects at the Stanford Medical School.

The purpose of the Project is to provide a computer system specifically designed for medical research. The ACME system is designed to act as a complement to currently existing facilities. It is assumed that a great deal of data-processing still will be done using the facilities of the Computation Center. By providing data storage and analysis facilities for the small computers, their capabilities will be greatly enhanced. In addition, by providing central signal-processing equipment, the system will support researchers that do not have their own computers.

A unique aspect of the Project is that it is a research project in support of research projects. In order to provide the medical school with the computational facilities necessary to do research, much development must be done by the project. It is planned that the ACME system will always be in a state of flux. As the more general-purpose computer systems are able to provide a service that is also being provided by ACME, that service will be dropped by ACME in favor of the other systems. At the time a need is found for a service not then available, that need will be met be extensions to the system. Thus, it is planned that the ACME system will be constantly devoted to the new and untried areas of computation, and in this way also will be a complement to existing facilities.

Within the ACME project, there is equal emphasis on hardware and software development. Hardware development is mainly concerned with interfacing the ACME system with the users and their experiments. Software development is concerned with providing non-computer oriented researchers with the tools necessary to do their work in a rapid and convenient manner. There has been a great deal of effort expended in developing a hardware/software complex where the hardware and software closely match both each other and the user and his experiment. Much effort has also been devoted to developing more suitable, and often less expensive, alternatives to manufacturer supplied hardware and software.

Both in terms of size and budget, the ACME Project is modest compared to many. But because of the fact that its goal is also relatively modest, that of providing a specialized set of services to a small and quite homogeneous set of users, it has already made a great deal of progress toward achieving that goal.

ACME system

The main purpose of the ACME system is the acquisition, analysis, storage, and retrieval of medical research data. In addition, there will be writing and debugging of programs necessary to support these activities. In the light of these requirements, it was decided that the most reasonable mode of operation for the ACME system would be time sharing; with emphasis on real-time data acquisition, and data storage and retrieval. This is an area where equipment that is currently available is very weak. Only very large specialized systems, mainly in military and space exploration environments, have achieved operational status. In order to make program writing and debugging as easy as possible it was decided that a compiler for a simple, yet relatively powerful programming language, would be written specifically for the ACME system.

It should be noted that the design criteria for the ACME system are relatively different from those of most other time sharing systems. Because of the demands of real time data acquisition, emphasis is not on giving fair and equitable service to a large group of users. Rather, it is to give high performance service to a relatively small group of users, while also answering the lighter demands of on-line program creation and debugging. Because of the demands of real-time operation, it is often better to refuse service to a user (telling him to try again later) than to offer a service that is degraded beyond usefulness to him or others using the system. Special provisions in the hardware and software have been made to reflect this philosophy.

Hardware for the ACME system

A general sketch of the ACME system is shown in Figure 1. The central processor for the ACME system is an IBM/360-50. This was chosen for several reasons. First, it is supported by a large amount of IBM-supplied software. Inasmuch as is possible, this software is used in preference to expending the effort to create similar software. Second, it is supported by a large variety of peripheral units, both mass-storage and input/output. Third, it provides a large measure of computing power and I/O versatility for a relatively low cost. Fourth, it provides upward-and-downward-compatibility with a broad line of computers, so that the system can be easily up-graded or down-graded to meet future conditions. Finally, it is highly compatible with the 360-67 at the Computation Center.

In order to provide real-time capability in a straightforward manner, it is necessary that the user program and data areas can be accessed very rapidly in a random manner. Core memory is the only device satisfying this requirement.

Estimates of the amount of memory required led to the following balance of the core memory versus number of users.

Assumptions:

(1) A control system of 7090 size.
(2) A resident compiler, library and input/output system, requiring 3 times the available memory of a 7090.
(3) User problem size distribution as on current large machines (i.e., IBM 7090's etc.) leading to an average of 14,000 words. See Figure 2.
(4) Program writing and translation will occupy the system 25% of the time.

With these assumptions:

$$\text{Memory required} = 6000 + 3 \times 26000 + n \times .75 \times 14000 + n \times .25 \times 6000$$
$$\text{or } 84000 + n \times 12000$$

This meant that a balance could be achieved at 15 users and 264000 words of memory.

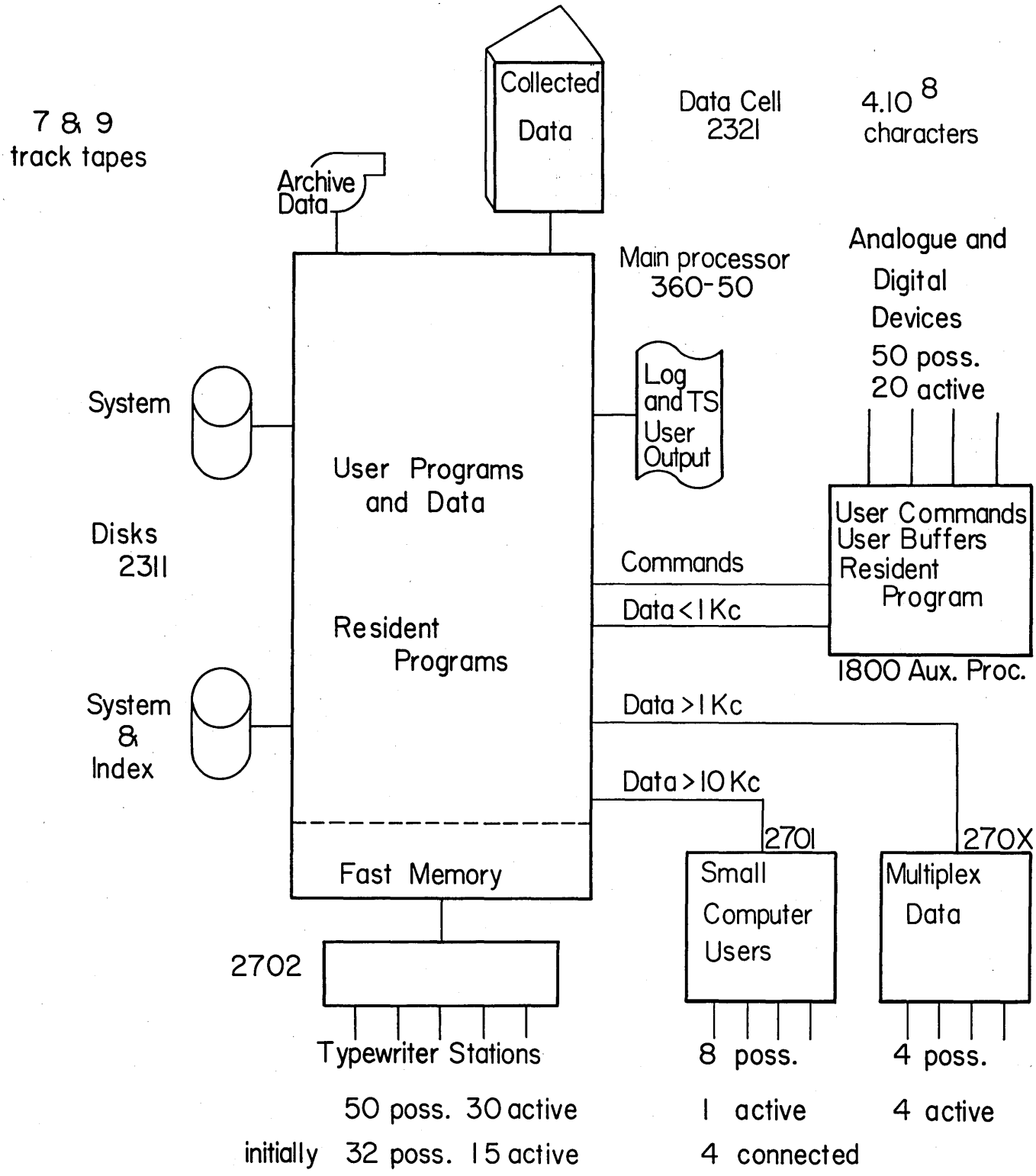
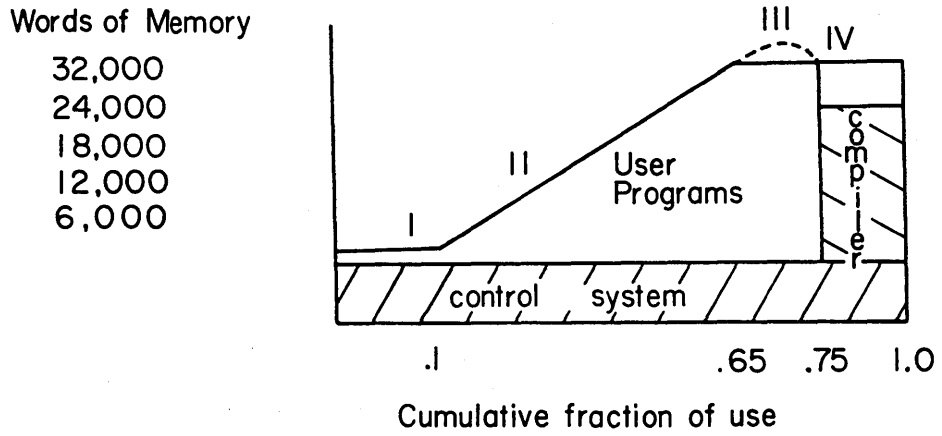


Figure 1—Sketch of ACME system



NOTES:

I. This region represents users using facilities on data stored in files, or doing desk calculator level work.

II. This region represents the normal computing workload. Much of this is used for data array storage. Declared arrays are generally only partially filled, leading to the apparent steepness of the curve.

III. Some percentage of users find that they tend to exceed the capabilities of the system and operate just below the maximum.

IV. During translation, program text and symbol tables very rarely exceed 6000 words.

Figure 2—User program size distribution

In order to get this amount of memory economically, it was decided that a minimum amount of relatively expensive fast memory would be obtained. The fast memory is used for residence of the 360 operating system and the most frequently used portions of the ACME software.

The major portion of the memory for the ACME system consists of a one million byte Large-Capacity Storage unit. This is attached to the central processor, and is addressable contiguously to the fast memory. Although its cycle time is much slower (8 μ sec vs. 2 μ sec), its cost per bit is about one-fourth of the central processor memory. Furthermore, the processor of the Model 50 can do some parts of a process without core references, and tests have indicated a factor at 2.3 performance degradation. In this way, enough memory could be obtained to keep the entire ACME system and all active user programs core-resident at all times. This is extremely important to the philosophy of providing high-performance service and is possible because of the small number of users. Because an active program is always core-resident, it is always executable, and hence can respond rapidly to real-time demands. Because all active programs are core-resident, memory allocation problems are minimized and such techniques as program relocation become unnecessary. The lack of memory swapping, paging, and concomitant problems such as special I/O buffering, greatly reduces system overhead.

Mass storage for the ACME system consists of three hierarchical levels. The lowest level is magnetic tape, which will be used for archival purposes. One 7-track unit is provided for compatibility with the 7090 and B-5500. One 9-track unit is provided, because of its higher performance and because it is necessary for generating the 360 operating system.

The next level of mass storage is the 2321 data cell. It has a storage capacity of 400 million bytes, with an average access time of 600 ms. This is the main storage device for the ACME system. On it, all user source programs and data are stored. Its capacity is large enough that it is expected that all programs and data will be permanently resident in the data cell, with no need to dump or reload from tape, except for back-up purposes. Special programming techniques have been developed to minimize the effect of its relatively slow access rate and to take advantage of its large capacity.

Another level of mass storage is the 2311 disc storage drive, of which there are only two in the ACME system. No user storage is provided here. One disc drive is used to store the non-core-resident portions of the 360 operating system. The other disc drive is used for two purposes. First, all of the ACME software is stored there, for initial loading when the system is started. The second, and most important, function of the drive is to store indices to information on the data cell. Whenever a user file on the data cell is opened, its location is determined from a catalog. An index to all records in the file is then moved from the data cell to the disc. Subsequent references to records in the file are then made using the disc-resident index.

Figure 3—Terminal indicator panel

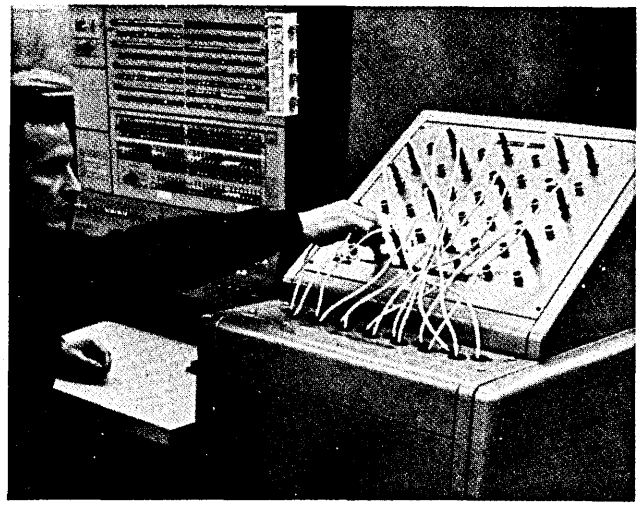


Figure 4—Switchboard

When the file is closed, the up-dated index is copied back from the disc to the data cell.

Each user of the ACME system has an IBM 2741 terminal. This is the device used to input programs, debug them, and control their execution. Since it is basically a modified Selectric typewriter, it has the advantage of being familiar and easy to use for non-computer oriented users. ACME has modified its 2741 by the addition of a 4 light indicator panel (Figure 3). Lights on the indicator panel are controlled by non-printing characters transmitted to the 2741. There is one light that reads, "ACME IS ON." It is driven by the transmission control signals to indicate that the system is operational. Another light reads, "YOU ARE ON." It is pulsed at a rate proportional to the amount of computing time the user is getting. The flicker rate thus indicates the performance of the system with respect to that user. Another light reads, "WAITING FOR YOU." It is on whenever the system is expecting input from the user. The final light reads, "SPECIAL RUN ON." It is on whenever a high demand, real-time data transmission process is active. It indicates that severely degraded terminal performance can be expected.

Most of the 2741's are connected directly by cables to the ACME system. Because of the fact that most of the 2741's are within 2000 feet of the system, no intervening cable drivers are necessary. The cables terminate in a switchboard-like patch panel (Figure 4). When a user wants to use his terminal, he telephones a computer operator, who connects his terminal to the system via the patch panel. Initially, there are 32 terminals with the possibility of 15 active at one time. Because of this low ratio, and the fact that terminal sessions are expected to be lengthy, manual switching is feasible at a great reduction in cost over other modes of operation. Communication with the 2741's is processed by a 2702 communication multiplexor.

The need for other than typewriter output devices has been obvious for a long time. But the cost of commercial devices in a medical environment is such that they are scarcely defensible in comparison to other medical aids.

However, two Sanders Associates character-oriented CRT displays have been ordered as experimental adjuncts to the ACME system. These will be used for development in areas of text editing and information retrieval. Because of the high data rate necessary to support these devices, they will be connected to the ACME system via a 2701 with a parallel data adapter. The interface to the 2701 is being supplied by Sanders Associates.

A special CRT display has been designed and built by the ACME Project for the input/output of graphical data (Figure 5). It is driven by vector-drawing logic and a core refresh memory. This unit is capable of displaying 2046 vectors simultaneously. It was built from integrated circuits, and the component cost, including the memory, was about $8,500. It will be connected to the ACME system via another parallel data adapter on the 2701.

A need still exists for a silent hard copy device for developing data distribution to hospital wards.

For the processing of user analog and slow (less than 1 Kc; i.e., 1000 samples per second) digital data an IBM 1800 computer is used. The 1800 is a small process control computer with a large complement of signal processing attachments. It will be used to do analog-to-digital and digital-to-analog conversions, as well as some primitive signal processing such as smooth-

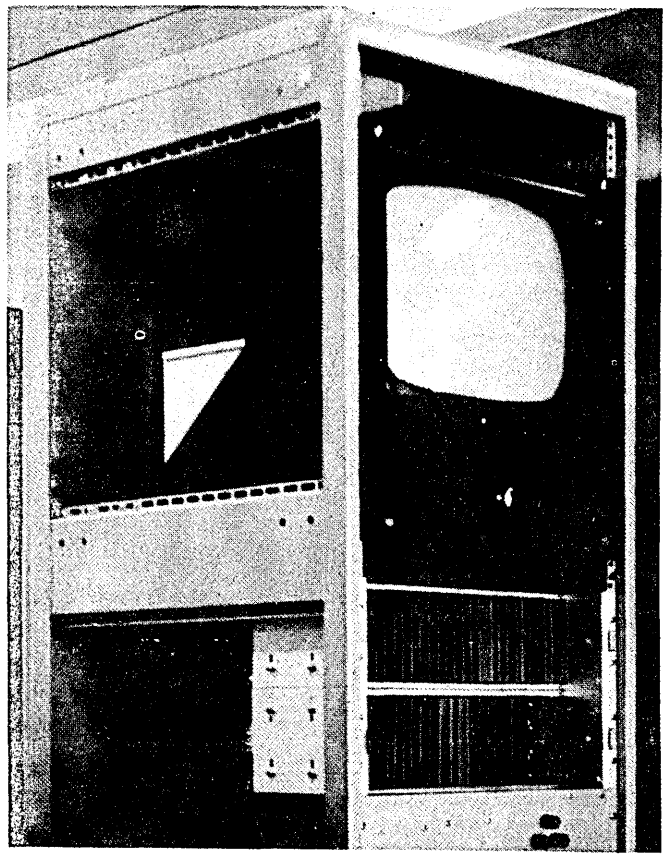


Figure 5—Prototype of graphical display

ing and validity checking. It is connected to the central processor via a high speed data channel so that it behaves essentially like an input/output device as far as the system is concerned. When relatively low bandwidth analog signals are transmitted, and no more than 8 bits of accuracy is needed, analog-digital conversion is done on the 1800. For more demanding signal processing, the analog-digital conversion has to be done in the laboratory and digital data is transmitted to the 1800. There are currently 32 analog-to-digital inputs, 8 digital-to-analog converters, 20 digital inputs, 12 digital outputs, and 80 process interrupt lines on the 1800. The timing of the data acquisition and distribution is user controlled through the process interrupt lines so that no synchronization of multiple experiment data rates is required. All analog and digital transmission equipment was designed and built by the ACME Project. One of the most interesting pieces of analog transmission equipment is an FM analog transmitter and receiver that is magnetically and acoustically coupled to an ordinary telephone. This will be used to process analog signals over distance or where no cables have been pulled.

For transmitting digital data in the speed range between 1 and 10 kc, a special multiplexor has been designed and constructed by IBM for the ACME Project. For lack of a better name, it is called the 270X. It is capable of multiplexing 8-bit digital transmission over cables between the ACME system and up to 16 remote units. Each remote unit is called a 270Y. There are currently four 270Y's. The 270Y's are designed for laboratory use, and are rack mountable. Each has binding posts for 8-bits of input and output, and for timing and control signals. Each has a push-button to terminate a transmission, and a signal line to cause a central processor attention interrupt. In addition, each 270Y has a built-in variable frequency oscillator to control sampling rate, or the sampling rate may be controlled via an external line. In addition to the local sampling-rate oscillator in each 270Y, there is a program-selectable oscillator in the 270X whose rate is adjusted to drive a digital incremental plotter. Thus, for example, a typical use of the 270X-270Y might be as follows. The inputs would be connected, via a suitable analog-to-digital converter, to an instrument such as a gas chromatograph. The sampling rate would be determined by the setting of the local oscillator on the 270Y. The outputs would be directly connected to an incremental plotter. A trial would be run, the data analyzed by a program that the user has written, and the results plotted immediately at the experimenters station. It takes little imagination to see the potential for this mode of operation.

It is expected that data rates in excess of 10 kc will be generated only by the CRT displays and the small computers. These will be connected to a 2701 with four parallel data adapters. Each parallel data adapter has a data path 16 bits wide, and is capable of sustaining speeds in excess of any demand now foreseen. Because of this high data rate, the 2701 has been connected to a central processor selector channel, which can sustain a high data rate with lower interference to computation. But since the selector channel can control only one transmission at a time, there must be a limit to the amount of time each transmission takes. Thus, it was decided that the normal mode for high data rate transmission (i.e., over 10 kc) will be short, very high speed bursts of data (i.e., 20 kc or higher). To connect the small computers to the 2701, a special interface was designed. The interface transmits data in parallel over cables on a demand/response basis. Provision has been made for remote computers to gain the attention of the central processor via an attention interrupt. In addition, a 25 ms. "dead-man" timer in the interface limits the time any single burst of data transmission can take.

The remaining hardware consists of unit record equipment on the 360 and 1800. This will be used

mainly for batch-mode operation during software development and checkout. In addition, during the early stages of time-shared operation the 360 printer is being used to keep a log of all transmissions to and from terminals. This should greatly facilitate the detection of problems that develop in using the system, both from the point of view of user difficulties and the detection of bugs in the software.

Software for the ACME system

The software for the ACME system will be divided into the following categories for purposes of discussion:

1. The 360 Operating System
2. The ACME compiler
3. Resource allocation (including time-slicing)
4. Data-file management
5. Terminal input/output
6. Real-time input/output
7. Library subroutines

One of the early, and significant, decisions in the design of the ACME system was the decision to use manufacturer supplied software whenever this was feasible. Thus, the entire ACME software system was designed to run as a single job under Operating System/360. During the operation of the ACME system, OS/360 provides such services as low-level input/output management and memory allocation, dynamic subroutine loading, and interval timing. This mode of operation has two significant advantages. First, a great deal of highly specialized programming can be avoided. Second, most of the remaining programs are machine independent, and hence can be written in a machine-independent manner. In fact, most of the ACME software has been written in FORTRAN. It was found that the IBM H-level FORTRAN compiler is capable of producing very efficient code and thus there was virtually no advantage in not using this machine-independent language. The few routines that were written at an assembly language level were mostly for such machine-dependent operations as character and bit-manipulation and for communication with the Operating System and for machine code skeletons.

Almost no modifications were made to OS/360. The few exceptions were in areas where the operating system provides for user-supplied modifications and these modifications were relatively straight forward Hence, almost anyone with a similarly configured 360 should be able to use the ACME software with a minimum of effort. In fact, with the multi-programming versions of OS/360, the ACME software is useable concurrently with other modes of operation.

The ACME compiler will be discussed fully in a future paper, and hence will be only briefly discussed here. The compiler is for a subset of PL/1 that includes many of the most useful features of the language. It is incremental; that is, it compiles one statement at a time and a program is always capable of execution. It compiles all the way to machine language, and thus produces relatively efficiently executing code. All system-user communication is processed by the compiler, hence the system command language is a subset of the compiler language, with identical syntax. Also included in the compiler language are text-editing functions for modification of program texts. For such processes as input/output, subscript range checking, and the computation of mathematical functions such as SIN, COS, and standard statistical procedures, the compiler generates calls to a resident library of subroutines. Some of these subroutines are also written in FORTRAN, and in fact use the IBM-supplied FORTRAN subroutine library for such things as input/output formatting.

Resource allocation consists mainly of memory allocation and time-sharing. Memory is allocated to users in 4048 byte (1024 word) quantities called pages. When a user has logged in he is assigned one such page in which his file names, etc., are kept. For programming two more pages will be assigned for program and data storage. More pages are assigned as they are needed during the compilation of his program. Memory for object-program arrays is not allocated until each array is referenced during execution. Hence there is some saving in memory during the time a user is creating his program. It is not necessary that the pages for a user program be contiguous in memory. Hence the problems of memory allocation are greatly simplified.

Because most of the ACME software was compiled under H-level FORTRAN, which is not capable of producing re-entrant code, allocation of time to users is not done in the usual manner. Instead of switching from one user to another at the end of some arbitrary time interval, switching is done only at so-called "re-entrant points." A re-entrant point is defined as a point at which:

1. All required information concerning the current user is located in storage peculiar to that user.
2. The next operation to be executed on behalf of the user is a call to a subroutine which will not return to the calling routine.

These two conditions are sufficient to insure a somewhat limited, but nevertheless adequate form of re-entrant programming. The scheme relies on the fact that the H-level FORTRAN compiler generates a prologue which is always executed upon entry to a subroutine. This prologue initializes information internal

to the subroutine, but peculiar to the current invocation of the subroutine.

The necessity of limiting switching to only certain points leads to the interesting situation in which the current user graciously yields his control of the machine, rather than having it wrested from him. However, proper etiquette in this regard is assured in several ways.

1. When a user is compiling a program, the structure of the compiler assures a reasonable discipline for user switching.
2. All requests for input/output activity require a yield.
3. A check is built into the code generated by the compiler at each GO TO and END statement for the end of a time interval. A yield will result if the time interval has elapsed.

In the process of yielding, a "resume routine" is indicated. This routine is entered when the user is next given control. Yields that accompany input/output requests generally specify a wait until the input activity is completed, due to the interactive nature of most system use. Users currently are served in strict order, on a round-robin basis. There is some probability that a priority scheme may be introduced later, if experience indicates that high data-rate experiments cannot be served with the round-robin scheme.

Because one of the major purposes of the ACME system is information storage and retrieval, special data management procedures have been designed to facilitate these operations. User data, and programs, are stored in data sets on the data cell. Data sets are ordinarily catalogued by user name, then by a project name, and finally by a user assigned data set name. When a data set name is mentioned in a program, it is automatically qualified by the user name and project name supplied at log-on time. This automatic qualification is overridden by explicit qualification of the form USERNAME.-PROJECTNAME.DATASETNAME. This method of data set naming is essentially identical to that used by OS/360, although OS/360 cataloguing procedures are not used. In addition, certain public data sets will be available. Among these will be a set of standard user-oriented programs.

Within the data sets, data is stored in the form of records. Records may be stored sequentially or randomly and may be retrieved sequentially or randomly. Records may be of arbitrary length. Data items are stored and retrieved from records by name. Hence, data items may be retrieved in an order different from the one in which they were stored. Moreover, fewer data items may be retrieved than are stored in a given record. This mode of operation is unlike that used in most current programming systems, where item order, not name, is significant. It is felt that the mode chosen, although somewhat less flexible, is much more in line with the thinking of the non-computer oriented personnel who will be the prime users of the ACME system.

The terminal input/output procedures used by the ACME system were also designed by the ACME Project. Output from the system is generally in the form of a message and a prompt. The message portion is the result of the last operation performed and the prompt portion is used to indicate what should be done next. A question mark (?) is used to indicate the end of the output. For example, during the log-on procedure the output from the system is the prompt NAME? The user then supplies his name. If the name is not acceptable, the system types an error message and re-prompts NAME? During compilation, the system prompts the line number of the next line when compilation of the preceding line is successful. If the compilation of the preceding line is not successful, the compiler supplies an error message and re-prompts the previous line number. Messages are generally quite long, so that a maximum amount of information can be conveyed. If a user recognizes a message and does not want to see it again in its entirety, he pushes the ATTENTION button on his terminal, which causes an ellipsis (. . .) to be typed and the rest of the message to be skipped.

The prompt portion of a message can usually not be ignored because it remains in a buffer as part of the user's next input. All prompts are recognizable to the compiler as commands, with the question mark treated as a blank. Hence, if PROJECT? is prompted and the user types DOGWEIGHTS, the input to the system is *PROJECT* DOGWEIGHTS. In fact, the system forgets what it has typed out; it only looks at the next input line to decide what to do next. If the user wishes to ignore a prompt, he may back-space over it, which causes it to be deleted, or he may push the ATTENTION key which causes the current input line to be ignored and a prompt of ? to be typed. Similarly, in the case of some syntax errors, the compiler merely prompts a corrected version of the statement. If the user wants to use the corrected version, he merely types carriage return, which causes the prompt to be used as input.

Real time input/output was also designed to maximize user convenience. With the exception of the 270X, which is still in the developmental stage, all real-time input/output is generated by satellite computers, either the ACME 1800 or the remote small computers. A communication protocol has been established for all transmission between computers. The protocol is designed

mainly for batch-mode operation during software development and checkout. In addition, during the early stages of time-shared operation the 360 printer is being used to keep a log of all transmissions to and from terminals. This should greatly facilitate the detection of problems that develop in using the system, both from the point of view of user difficulties and the detection of bugs in the software.

Software for the ACME system

The software for the ACME system will be divided into the following categories for purposes of discussion:

1. The 360 Operating System
2. The ACME compiler
3. Resource allocation (including time-slicing)
4. Data-file management
5. Terminal input/output
6. Real-time input/output
7. Library subroutines

One of the early, and significant, decisions in the design of the ACME system was the decision to use manufacturer supplied software whenever this was feasible. Thus, the entire ACME software system was designed to run as a single job under Operating System/360. During the operation of the ACME system, OS/360 provides such services as low-level input/output management and memory allocation, dynamic subroutine loading, and interval timing. This mode of operation has two significant advantages. First, a great deal of highly specialized programming can be avoided. Second, most of the remaining programs are machine independent, and hence can be written in a machine-independent manner. In fact, most of the ACME software has been written in FORTRAN. It was found that the IBM H-level FORTRAN compiler is capable of producing very efficient code and thus there was virtually no advantage in not using this machine-independent language. The few routines that were written at an assembly language level were mostly for such machine-dependent operations as character and bit-manipulation and for communication with the Operating System and for machine code skeletons.

Almost no modifications were made to OS/360. The few exceptions were in areas where the operating system provides for user-supplied modifications and these modifications were relatively straight forward Hence, almost anyone with a similarly configured 360 should be able to use the ACME software with a minimum of effort. In fact, with the multi-programming versions of OS/360, the ACME software is useable concurrently with other modes of operation.

The ACME compiler will be discussed fully in a future paper, and hence will be only briefly discussed here. The compiler is for a subset of PL/1 that includes many of the most useful features of the language. It is incremental; that is, it compiles one statement at a time and a program is always capable of execution. It compiles all the way to machine language, and thus produces relatively efficiently executing code. All system-user communication is processed by the compiler, hence the system command language is a subset of the compiler language, with identical syntax. Also included in the compiler language are text-editing functions for modification of program texts. For such processes as input/output, subscript range checking, and the computation of mathematical functions such as SIN, COS, and standard statistical procedures, the compiler generates calls to a resident library of subroutines. Some of these subroutines are also written in FORTRAN, and in fact use the IBM-supplied FORTRAN subroutine library for such things as input/output formatting.

Resource allocation consists mainly of memory allocation and time-sharing. Memory is allocated to users in 4048 byte (1024 word) quantities called pages. When a user has logged in he is assigned one such page in which his file names, etc., are kept. For programming two more pages will be assigned for program and data storage. More pages are assigned as they are needed during the compilation of his program. Memory for object-program arrays is not allocated until each array is referenced during execution. Hence there is some saving in memory during the time a user is creating his program. It is not necessary that the pages for a user program be contiguous in memory. Hence the problems of memory allocation are greatly simplified.

Because most of the ACME software was compiled under H-level FORTRAN, which is not capable of producing re-entrant code, allocation of time to users is not done in the usual manner. Instead of switching from one user to another at the end of some arbitrary time interval, switching is done only at so-called "re-entrant points." A re-entrant point is defined as a point at which:

1. All required information concerning the current user is located in storage peculiar to that user.
2. The next operation to be executed on behalf of the user is a call to a subroutine which will not return to the calling routine.

These two conditions are sufficient to insure a somewhat limited, but nevertheless adequate form of re-entrant programming. The scheme relies on the fact that the H-level FORTRAN compiler generates a prologue which is always executed upon entry to a subroutine. This prologue initializes information internal

to the subroutine, but peculiar to the current invocation of the subroutine.

The necessity of limiting switching to only certain points leads to the interesting situation in which the current user graciously yields his control of the machine, rather than having it wrested from him. However, proper etiquette in this regard is assured in several ways.

1. When a user is compiling a program, the structure of the compiler assures a reasonable discipline for user switching.
2. All requests for input/output activity require a yield.
3. A check is built into the code generated by the compiler at each GO TO and END statement for the end of a time interval. A yield will result if the time interval has elapsed.

In the process of yielding, a "resume routine" is indicated. This routine is entered when the user is next given control. Yields that accompany input/output requests generally specify a wait until the input activity is completed, due to the interactive nature of most system use. Users currently are served in strict order, on a round-robin basis. There is some probability that a priority scheme may be introduced later, if experience indicates that high data-rate experiments cannot be served with the round-robin scheme.

Because one of the major purposes of the ACME system is information storage and retrieval, special data management procedures have been designed to facilitate these operations. User data, and programs, are stored in data sets on the data cell. Data sets are ordinarily catalogued by user name, then by a project name, and finally by a user assigned data set name. When a data set name is mentioned in a program, it is automatically qualified by the user name and project name supplied at log-on time. This automatic qualification is overridden by explicit qualification of the form USERNAME.-PROJECTNAME.DATASETNAME. This method of data set naming is essentially identical to that used by OS/360, although OS/360 cataloguing procedures are not used. In addition, certain public data sets will be available. Among these will be a set of standard user-oriented programs.

Within the data sets, data is stored in the form of records. Records may be stored sequentially or randomly and may be retrieved sequentially or randomly. Records may be of arbitrary length. Data items are stored and retrieved from records by name. Hence, data items may be retrieved in an order different from the one in which they were stored. Moreover, fewer data items may be retrieved than are stored in a given record. This mode of operation is unlike that used in most current programming systems, where item order, not name, is significant. It is felt that the mode chosen, although somewhat less flexible, is much more in line with the thinking of the non-computer oriented personnel who will be the prime users of the ACME system.

The terminal input/output procedures used by the ACME system were also designed by the ACME Project. Output from the system is generally in the form of a message and a prompt. The message portion is the result of the last operation performed and the prompt portion is used to indicate what should be done next. A question mark (?) is used to indicate the end of the output. For example, during the log-on procedure the output from the system is the prompt NAME? The user then supplies his name. If the name is not acceptable, the system types an error message and re-prompts NAME? During compilation, the system prompts the line number of the next line when compilation of the preceding line is successful. If the compilation of the preceding line is not successful, the compiler supplies an error message and re-prompts the previous line number. Messages are generally quite long, so that a maximum amount of information can be conveyed. If a user recognizes a message and does not want to see it again in its entirety, he pushes the ATTENTION button on his terminal, which causes an ellipsis (. . .) to be typed and the rest of the message to be skipped.

The prompt portion of a message can usually not be ignored because it remains in a buffer as part of the user's next input. All prompts are recognizable to the compiler as commands, with the question mark treated as a blank. Hence, if PROJECT? is prompted and the user types DOGWEIGHTS, the input to the system is *PROJECT* DOGWEIGHTS. In fact, the system forgets what it has typed out; it only looks at the next input line to decide what to do next. If the user wishes to ignore a prompt, he may back-space over it, which causes it to be deleted, or he may push the ATTENTION key which causes the current input line to be ignored and a prompt of ? to be typed. Similarly, in the case of some syntax errors, the compiler merely prompts a corrected version of the statement. If the user wants to use the corrected version, he merely types carriage return, which causes the prompt to be used as input.

Real time input/output was also designed to maximize user convenience. With the exception of the 270X, which is still in the developmental stage, all real-time input/output is generated by satellite computers, either the ACME 1800 or the remote small computers. A communication protocol has been established for all transmission between computers. The protocol is designed

around the concept of a conversation between the main computer and a satellite computer that can be initiated by either party. Once initiated, the conversation continues until there is no more data to be transferred in either direction. Because some satellite computers, such as the 1800, may be processing several different sets of input/output concurrently, provisions have been made for several concurrent conversations between the same pair of computers.

As far as the user is concerned, real-time input/output is programmed in exactly the same way as data-file input/output. Each real-time input/output path is treated as a data set by the ACME system. The data sets are catalogued in the same way as data-file data sets and accessed by the same set of commands. Data set attributes, catalogued with each data set, allow the software to distinguish between real-time and data-file sets. Conversion is automatically done by the ACME software to provide compatability between satellite computer data formats and ACME system data format.

The subroutine library consists of commonly used subroutines for such operations as statistical analysis, graph plotting, etc. Although some subroutines may be written in the ACME compiler language, the majority will probabily remain in FORTRAN or assembly language, due to the higher object program efficiency that results. The most commonly used routines will always be core-resident, with direct linkage provided by the ACME compiler. These are not re-enterable, so their execution time must be small. Less frequently used subroutines, or subroutines with long execution time, will be dynamically loaded and assigned to individual users. When these routines are no longer needed, the memory they occupy will be released and made available to other users.

CONCLUSION

As of July, 1967, all of the hardware described in this paper is operational, with the exception of the Sanders Associates displays which have not been delivered yet. Connection has been established between the 1800 and two research laboratories. One small computer has been connected to the system. The ACME compiler is almost complete. Timing tests have indicated that its object code efficiency compares favorably with that of similar compilers. The resource allocation software is complete and allows time-shared use of the system. The terminal input/output routines also have been completed. The data-file management routines allow only program storage, due to a delay in IBM software support for the data cell. The real-time input/output routines are in an advanced stage of development, and currently allow primitive real-time input/output. The subroutine library is partially complete, with work continuing to expand its scope.

It is hoped that this paper will provide encouragement to those who believe that a successful time-sharing system is possible and is realizable by an organization with limited resources.

At a time when many highly-touted time-sharing systems are proving to be less than successful, the ACME Project is quite proud of its accomplishments. If there is any lesson to be learned, it is that a small group, with specific and well defined goals and a highly cooperative user community, can quickly and effectively provide a needed service to that community.

ACKNOWLEDGMENTS

The planning work was sponsored by a Josiah Macy Foundation planning grant and further funding has been granted from NIH (Grant No. FR-00311) and Macy Foundation. Much credit for these ideas and procedures goes to other computer installations and other people, notably project MAC at M.I.T., MEDLAB at the Latter-Day Saints Hospital in Salt Lake City, the Computer Center and ARPA project at the University of California, Berkeley; University of California San Francisco Medical School; U.C.L.A. Health Sciences, etc., and of course the Computation Center and the Computer Science Department of Stanford itself.

REFERENCES

1 H D HUSKEY W WATTENBURG
A basic compiler for arithmetic expressions
Communication of the ACM January 1961

2 W KEESE H D HUSKEY
An algorithm for the translation of Algol statements
Proceedings of the IFIP Congress 1962

3 F J CORBATO et al
The compatible time-sharing system
MIT Press 1963

4 G WIEDERHOLD
A proposal for a simple system allowing direct access to the computer
Internal Paper Berkeley Computation Center 15 May 1963

5 J H SALTZER
TYPSET and RUNOFF
Memorandum editor, MAC Memo 193-2 11 January 1965

6 G WIEDERHOLD
Student, a fast FORTRAN IV compiler
Internal documentation Berkeley Computation Center 1965

7 J SHARP C GRAM B PANZL
Student language manual, The language and the processor
Department of Electrical Eng and Computer Science Berkeley 1 March 1966

8 H BERG et al
Report of the SHARE Advanced Language Development Committee 1 March 1964

9 A J SCHERR
Time sharing measurement
Datamation February 1966

10 G WIEDERHOLD
A summary of the ACME system
Proceedings of the ONR Computer and Psychobiology Conference Monterey 17 May 1966

11 G Y BREITBARD et al
ACME notes
Internal documentation Stanford Computation Center ACME Facility November 1965 to present

12 V WIEDERHOLD
How to use PL/ACME
Document No 80-50-00 Stanford Computation Center 15 July 1967

A panel discussion

The impact of new technology on the analog hybrid art–I

G. A. BEKEY, Chairman
University of Southern California
Los Angeles, California

Present day hybrid systems are characterized by increasingly sophisticated software requirements. The first attempts at creation of useful analog-digital computer systems were faced with a multitude of hardware problems associated with communication between discrete and sequential machines on the one hand and continuous and parallel machines on the other. Now, however, since the hardware marriage has been successfully consumated, a multitude of software problems remain. This panel will concentrate on the most important of these problems. Position papers by each of the four panelists are presented below.

Hybrid executive and problem control software

by E. HARTSFIELD
TRW Systems
Redondo Beach, California

Hybrid executive and problem control software provides features quite analogous to those provided in Monitor systems associated with stacked job oriented data processing centers. The major difference is that a hybrid executive routine stresses *execution* time utility as opposed to the compilation and assembly time utility emphasis of stacked-job Monitors. If one accepts the proposition that hybrid simulation is an extension of analog simulation, then the need for a software system that facilitates communication between the user and his program is brought clearly into focus.

Unlike a data-center mode of operation, where the user submits his job to an operations desk and does not receive results (good or bad) for several hours, a hybrid user operates on the same program for several hours at a time. In addition, there are a number of operations common to a vast majority of hybrid problems that lend themselves to standardization through an executive routine. This would include such operations as data input and output, bilateral problem control (i.e., the sequencing of the analog computer through modes from either the digital computer or the analog computer), and problem debugging procedures.

Hybrid executive software is aimed at solving these problems. There is a wide diversity of executive routine philosophies and implementations, reflecting, perhaps, the fact that no two hybrid installations are the same either in a systems sense or in terms of the modes operandi of the facility.

This presentation reviews the general state-of-the-art in hybrid executive software and some of the major considerations one faces when initiating the development of software of this type.

Diagnostic software for operation and maintenance of hybrid computers

by R. E. LORD
Comcor/Astrodata
Anaheim, California

Diagnostic software is based upon the premise that programmers occasionally make mistakes and that sometimes hardware behaves less than ideally. The major problems in producing good diagnostics are: (1) how does one find the errors and (2) once they have been found, what does one do about them. In the case of maintenance software, the problems are primarily in finding the errors. Once found, the course of action is fairly clear: report it. In the case of operational diagnostics, the course is not quite so clear. For those operations that occur during a time critical phase, one must examine whether there is time available for

complete diagnostics. or these cases, it is often better to build diagnostics into hardware than it is to rely on software. During non-critical phases of a problem, complete diagnostics can usually be performed and errors readily detected. The problems here are in determining what action to take. In general, batch processing takes the view that any error serious enough to diagnose for, is serious enough to cause an abort. The hybrid problem however, generally involves a high degree of man-machine interaction and hence, upon diagnosing an error, one should determine if the user can fix or quickly program around the error. If this is the case, then a complete report of the error should be made. Only as a last resort should an abort be initiated.

A large multi-console system for hybrid computations: software and operation

by C. K. BEDIENT
Lockheed Missiles and Space Company
Sunnyvale, California

In the first quarter of 1966, Lockheed Missiles and Space Company completed negotiation for a new hybrid computer system. This system contained four CI 5000 analog computers and two intracoms manufactured by Comcor, Inc. It also contained a 6400 digital computer system manufactured by Control Data Corporation. Astrodata, the parent company of Comcor, was given complete system responsibility including software development. This discussion traces this software development and the resulting operating system.

To implement this software development, Lockheed, Control Data, and Astrodata were each to supply programming support. To direct the programming effort, Astrodata retained Dr. Ralph Dames of Spectrodata.

The first part of the discussion reviews the features of the 6400 computer and the Chippewa operating system. It reviews the modifications required to this system to give it the features required to do hybrid computation. Tracing the design of the linkage equipment it is shown how features of the hardware and software are used to advantage to produce the final integrated software hardware package. This includes a description of interrupt processing, pattern input-output, and the effects of multiproblem usage.

The second part of the discussion shows the organization of a simple problem and demonstrates its operation on the system.

Other features of the system are briefly outlined, including:

1. Fortran Library
2. Display System
   a. Job Control
   b. Variable Display
   c. Source Modification
   d. Input/Output
   e. Preventive Maintenance
   f. Engineering Aids
   g. Hybrid Utility
   h. Datafile Display
3. Preventive Maintenance
4. Automatic Problem Verification

Simulation languages and the analog hybrid field

by JON C. STRAUSS
Carnegie Institute of Technology
Pittsburgh, Pennsylvania

Languages to facilitate the representation and simulation of continuous dynamic systems on digital computers are currently related to analog/hybrid computation mainly in the sense that they both are concerned with the same problem class (with the possible exception of real time simulation). There is, however, some indication that current work will lead to general hybrid programming languages. This brief position paper describes some of the current work in simulation languages, its relation to previous work, and its potential relation to the analog/hybrid field.

The early simulation languages were very much influenced by the outlook and objectives of their analog computer oriented designers; i.e., the work was motivated by the need for accurate dynamic check solutions. As digital computers become faster and less expensive, these analog oriented simulation languages were increasingly employed for stand alone digital simulation. The justification was generally based on the greater accuracy, accessibility, and result reproducibility of the digital computer coupled with a much smaller relative setup cost and time for one shot simulations.

Due primarily to the orientation of their designers, most of the early simulation languages provided for continuous system simulation by simulating the operational device characteristics of the analog computer; system simulations were programmed by, in effect, describing an analog computer wiring diagram and pot setting sheets to the computer. It was not until very recently that simulation language designers began to take advantage of the fact that the simulations are

being performed on a digital computer with capabilities that are just not present on the typical analog. The most recent and certainly the most carefully designed simulation language is CSSL (the Continuous System Simulation Language); it is the product of several years work of the SCI Simulation Software Committee. CSSL, although not presently implemented in its entirety, serves as an excellent basis for discussion of simulation languages, their relation to, and their ultimate effect on, the analog/hybrid simulation field.

CSSL was designed primarily as a digital simulation language but care was taken to provide for future upwards compatible expansion to a complete hybrid programming language. The main contribution of CSSL lies not in its detailed syntactical structure but rather in the underlying ideas that it presents. These include:

1. The explicit recognition of the different requirements of simulation in general; namely: model representation, programmed experimental control, interactive control/communication, and problem oriented operators to describe their problems.

2. The need for a flexible model representation scheme so that investigation in different fields can use the same language system with different sets of problem oriented operators to describe their problems.

3. The concept of programmable structure that permits and, in fact, encourages use of the same simulation system by users, and for problems, varying greatly in sophistication.

The question at hand is how does all this relate to the analog/hybrid field. The following observations are pertinent if somewhat controversial:

1. Simulation Languages and all digital simulation will continue to find greater application areas now considered to be the exclusive province of analog and hybrid computers. This trend will be accelerated by the development of graphic consoles and time shared programming techniques that will allow the user to achieve the same or perhaps even greater degree of communication with his problem that he now enjoys with analog computation.

2. As evidenced by the design features of CSSL, development of HSL, a dialect of CSSL, by EAI, and recent indications in the literature (e.g., R.T. Dames, Simulation, March 1967), simulation languages will find increased application as hybrid programming languages; i.e., they will be used to program the digital portion of a hybrid problem.

3. If the interest in, and need for, hybrid computation continues to increase, the current unpleasant task of getting a problem on the computer will be automated through development of hybrid computer analogs of APACHE. Such work could be readily mechanized in the general framework provided by CSSL.

A panel discussion

Information services and communications (computer utilities)

E. M. GRABBE, Chairman
TRW Systems
Los Angeles, California

With the continuing increase of computer installations utilizing some form of data communication, experience is being accumulated on the remote terminal operation of computer and information services for interactive usage, batch processing and other data retrieval services. Important factors in the future commercial development of such services are the availability and cost of communication circuits and terminal equipment. Interest has recently been focused on these areas by the Federal Communications Commission's inquiry into the "regulatory and policy problems presented by the interdependence of computers and communications service facilities."

The goals of the panel will be to clarify the interdisciplinary problems and relationships between computers and communications which, as one panel member stated, "have placed the computer and communication industries on a collision course." Specific areas of information services to be reviewed in relation to communications are: experience in operating hardware and software systems, interface requirements, system economics, communication circuits and public policy.

Position papers submitted by three of the panel members follow.

Time-shared information systems: market entry in search of a policy*

by MANLEY R. IRWIN
University of New Hampshire
Durham, New Hampshire

I. INTRODUCTION

Recent developments in computer hardware and programming now enable several users to share the stored information and logic capability of data processing machines. The subscriber, no longer adjacent the computer, may access the computer's logic and memory via telephone lines tied to the data center. Indeed, the combination of data centers, communication lines, and termnial equpiment form the elements of what some forecast as a new industry, time-shared services or what others call the computer or information utility. Services arising out of this industry includes bibliographic retrieval, stock quotation, hotel reservations, legal indexing, market reports, banking by phone, hospital information systems, common data files, program libraries, to mention a few. Such services soon promise to be international as well as domestic in operation and scope.

Last year, the Federal Communications Commission initiated an investigation into the policy implications of computer time-shared, operations.[1] The Commission seeks to determine its statutory obligations given what it terms the "growing interdependence" of computers and communications. This paper proposes to examine one of the issues posed by the Commission's investigation. Specifically, we will (1) state the regulatory issues as the Commission views them; (2) discuss the background events that prompted the inquiry; and (3) evaluate some of the competitive issues associated with time-shared computer services. We will conclude that the ground rules for market entry are at stake in the FCC's investigaiton.

II. The investigation

The Federal Communications Commission announced its computer inquiry on November 10, 1966. Some of the Commission's questions dealt with the adequacy of tariffs, rates, and customs practiced by the communication carrier industry. Other questions focused on the issue of privacy and telephone wiretapping incident to the growing concentration of information. The central if not the most controversial issue, however, settled on the question of the Commission's regulatory obligations. Specifically, the Commission asked:

> The circumstances, if any, under which any of the aforementioned services (data processing, special information services, message or circuit switching) should be deemed subject to any regulation pursuant to the provisions of Title II of the Communications Act.
>
> (1) when involving the use of communications facilities and services;
> (2) when furnished by an established communication common carrier;
> (3) when furnished by entities other than established communication common carriers.[2]

These questions may have caught some observers by surprise, particularly members of the computer industry. However, a review of the factors contributing to the inquiry suggests that the FCC may have had little choice in exposing these issues for public discussion. In fact, the Commission's attempt to distinguish between "data processing" and "communications" may prelude a more general policy search in the growing field of time-shared computer services.

III. Source of investigation

First, what prompted the investigation. Obviously the inquiry resulted from the interplay of many factors. However, four can be identified as being singularly important: (1) the Bunker-Ramo case, (2) the IBM letter, (3) dinersification efforts of Western Union, a domestic telegraph carrier, (4) diversification efforts of ITT Worldcom, an international telegraph carrier. The background of each follows.

A. The Bunker-Ramo case

The first issue, determining the content of a communication common carrier service, erupted in a service offering by the Bunker-Ramo Corporation. Bunker-Ramo sells a stock quotation service to brokers, bankers and insurance companies. Stock information is gathered from the main Exchange floor in New York, transmitted and stored in Bunker-Ramo's regional offices. Computers in these offices then distribute this information via telephone lines to the firm's customers throughout the country. By means of desk units supplied by the company, subscribers query the computer for a host of market security information. Bunker-Ramo, designating its service as Telequote III, operated an information retrieval system on a real-time or instant use basis.

Under this system, Bunker-Ramo provided its subscribers with stock quotation service, desk display units, and telephone communication circuits, all as a total package. Although Bunker-Ramo owned the former two units, it obviously did not own the latter. Communication lines were necessarily secured from the communication common carrier industry and then resold by Bunker-Ramo to its customers.

As a rule, the common carriers do not lease communication lines to firms who resell them. The carriers hold such activity subsidizes their competition, dilutes revenues, and ultimately translates into higher rates for their telephone or telegraph customers. The carriers argue, moreover, that the exchange or routing of messages among or between persons constitutes the essence of "communications" as conventionally defined. This function is obviously confined to regulated communication firms.

There are, however, exceptions to this policy. Reselling of communications is permitted under the carriers' authorized user tariff. A firm may resell circuits if the carriers determine that both the firm and its customers are engaged in the same business activity. Bunker-Ramo customers, stockbrokers, etc., qualified under the authorized user tariff. Thus, the firm acting as literally a broker between its customers and the carriers, sold leased communication facilities.[3]

The announcement of a new stock quotation service, Telequote IV, reopened not only the question of definition but also the question of circuit availability. In contrast to Telequote III, Telequote IV grafted electronic message switching to the computer's capability of data processing, message switching that permitted brokers to place, execute and confirm stock orders between their offices. The carriers agreed that the message switching element constituted a new and crucial element to Bunker-Ramo's service.

Specifically, the carriers held that the store and forwarding of information constituted a tariff violation. Stated differently, Bunker-Ramo's service embraced enough communications so as to infringe upon an activity reserved to regulated entities.[4] The implication was strong that if communications was Bunker-Ramo's intent then the firm ought to apply for a franchise of convenience and necessity. In the meantime, the denial of communication lines effectively foreclosed the operation of Bunker-Ramo's new service.

This impasse prompted Bunker-Ramo to issue a letter of complaint to the FCC. The firm granted that its message switching of administrative data might be interpreted as the transportation of communications. However,

> Such transportation rarely occurs as an incident to another primary business and although a charge is collected for the overall service, the incidental transportation is not supplied with the purpose to profit from whatever transportation service may thus be performed.[5]

In short, Bunker-Ramo invoked the primary business test.

To emphasize the incidental nature of Telequote IV's communications, Bunker-Ramo revealed a breakdown of Telequote IV; 84% was assigned to services of stock quotations; 10% assigned to orders and confirmations; 1% to market opinions, and 3% to customer accounting. The controversial segment, the routing of administrative data, consisted of some 26% of the entire stock quotation service.[6] Clearly, argued Bunker-Ramo, this small portion did not constitute communication activity in its generally accepted sense;

and clearly such activity should not condemn Bunker-Ramo's entire service as illegal.

Ultronics Systems Corporation, a competitive supplier of stock quotation service, volunteered its observations to the FCC. Ultronics submitted that the Telequote IV service embraced enough message switching or communications so as to indict the entire service as a regulated communication activity.[7] Interestingly enough, this position paralleled Western Union's thesis that communication rather than data processing was actually the dominant element of Bunker-Ramo's contemplated service.

In the end, the Telequote IV dispute did not require the Federal Communications Commission to promulgate a formal ruling. Subsequent negotiations between Bunker-Ramo and the Bell Telephone System (AT&T) found room for compromise. Bunker-Ramo agreed to eliminate the 2% administrative segment of its service; Bell agreed to lease circuits under its authorized user tariff. As a result, the policy issues latent with this dispute were merely postponed.

B. The IBM letter

The FCC's inquiry was also identified with an open letter submitted to the Federal Communications Commission by IBM.[8] The letter was both cautious and apprehensive. IBM began by drawing a sharp distinction between the terms "communication" and "data processing." Data processing or the transformation consisted of analyzing, classifying, correlating, sorting, calculating, summarizing, and producing records and reports. By contrast, communications embraced the transportation or transmission of information by wire or radio under the Communications Act of 1934.

Lest the Commission confuse the two, IBM observed:

> If intelligence is sent from A to a computer and transformed by the computer and the message then goes forth from the computer to B containing the transmitted intelligence, it is clear that there has not been transmittal of a message from A to B through a computer.[9]

The issue in the Bunker-Ramo case, however, was not solely one of determining the appropriate mix between data processing and communications. The parties generally agreed that Telequote IV embraced the electronic routing of message communications. The question, however, turned on the status of switching activity performed by a firm not licensed as a communication carrier. IBM suggested the primary business test as a guide to resolving any such determination. And if communications were indeed incidental, such activity did not qualify as regulated, limited exclusively to telephone or telegraph companies.

What about the reverse situation, the case when a regulated carrier engages in data processing? Obviously aware of Western Union's diversification into data processing services, IBM cited an FCC decision with respect to the telegraph company's flower service. Though part of its communication plant is so employed, Western Union does not file tariff schedules on the flower segment of the service. This policy, argued IBM, provided sound precedent for Western Union's or any carrier's move into unregulated, data processing services.[10]

In sum, IBM counseled the FCC to consider the primary business test when non-regulated firms find themselves engaged in regulated or message switching activities. By the same token, IBM submitted that data processing services supplied by regulated entities does not automatically redefine that activity as a regulated one.

C. Western Union's diversification

Since telephone or telegraph lines are by definition critical to remote data processing services, it was perhaps invitable that the communication carriers would find computer services an attractive source of market diversification. Indeed, Western Union's concept of a national information utility, a blending of computer and communication faciltes, epitomized that diversification.[11] To this end, the telegraph company announced the computerization of its Telex switching centers, the establishment of lata processing service centers, the announcement of specific data processing services—legal index, employment service, among others—and the introduction of a management information system available to commercial and government users.

Western Union's entry into information services originated with its government contracts. The Autodin contract, for example, consisted of a joint data processing and message switching system to the Defense Department. A similar service, the Advanced Record System, was also leased to the General Services Administration; a system whereby the telegraph company has been responsible for the automatic dial nationwide circuit switching system, three message switching centers, and the supply and maintenance of software and message switching capability. (The computers route multiple address messages, recognize priorities, and hold messages for specific delivery time.)

The seeds of controversy inherent in a regulated firm engaging in a non-regulated activity commenced with these government contracts. As early as August 1964, the FCC requested Western Union to justify why the data processing segment of the Advanced Record Sys-

tem, was excluded as a tariffed service. The Commission wrote:

> We are advised that you do not intend to include in your tariffs on file with the Commission charges for off-line processors furnished in connection with this system. It is requested that you advise us as to what you believe to be the legal justification for furnishing this item of equipment without having charges therefor on file in your tariffs.[12]

Later, the question of tariffing and contracting arose again with Western Union's announcement of its management information system for Dun & Bradstreet and Blue Cross.[13] Finally, last year the FCC urged the telegraph company once again to explain its policy with respect to its data processing services.[14]

Western Union's response attempted to place its new activities in perspective. The telegraph company reminded the Commission that it compted directly with firms whose computers were replacing Western Union's torn-tape switching systems. Rivalry between the carriers and computer firms, explained Western Union, was immediate and direct in the GSA Advanced Record System contract; and Western Union argued that had it tariffed its service, the telegraph company would have been placed in an untenable position vis-a-vis its competitors.

To illustrate the need for flexibility and hence tariff policy, Western Union recalled that its original bid to GSA had been high. A downward revision occurred through the cooperation of Univac's Division of Sperry Rand—a revision that resulted in a bid that ultimately secured the contract for the telegraph company. Western Union continued:

> GSA expressed concern that since the Western Union bid was of a tariffed nature, the quoted price was not firm for (a) tariffs would be necessarily filed months after the award of a contract, and (b) regulatory action would be in effect bring about a superseding of the contract. Western Union was therefore advised that it could have a contract only on the condition that a firm fixed price contract for five years could be offered, and had Western Union not acceded, GSA would have made a contract award to ITT.[15]

What was true for the government sector was equally true for the private sector. To meet market competition, Western Union responded that its data processing activities need not be subject to regulation despite the fact that the company itself was a regulated carrier.

Finally, Western Union elicited restrained enthusiasm from the primary business test as a policy guide to determining the communication activities of data processing firms. The primary business test, observed the telegraph company, threatened the very concept of regulation.

> Such companies as IBM would always be in the position of furnishing a part of the communication business on an incidental basis and hence, would never be subjected to regulation. The safeguards of the utility concept would thus be eroded.[16]

In specific reference to Telequote IV, Western Union was concerned that the element of direct communication among the subscribers would persist. Although the compromise between Bunker-Ramo and the Bell System may have been workable to Bunker-Ramo, it was less than satisfactory from Western Union's standpoint.[12]

D. ITT Worldcom's diversification

A third element that impelled the Commission's Notice of Inquiry was associated with developments on the international scene, the international communication carrier. Last year, ITT Worldcom, a subsidiary of International Telephone & Telegraph and an overseas record carrier, filed a tariff on a computer message switching service .The tariff, designated ARX (customer data re-transmission) excluded individual circuits and terminal apparatus. Rather, the subscriber was billed on the number of messages routed through ITT's computer switching system.

The tariff encountered opposition immediately.[18] Western Union, for example, objected that as an international carrier, ITT was diversifying into the domestic communication market competing with and eliminating Western Union's torn-tape relay switching systems, thus diluting needed revenues from the telegraph company. Western Union International Telegraph Company, an overseas record carrier, joined Collins Radio, a manufacturer of computer switching equipment, in voicing further objections to the ARX tariff. Western Union International asserted that the tariff was unclear and ambiguous; and Collins Radio suggested that the tariff was underpriced and hence, non-compensatory.[19]

ITT Worldcom acknowledged that some of its switching systems would be located within the United States. However, as long as its customers leased one private line circuit overseas, domestic switching was permitted under the Communications Act, Section 222. (The act defines an international carrier as one whose major portion of traffic and revenues is derived from international telegraph operations.)[20]

Significantly, ITT Worldcom asserted that computerized message switching was an activity subject

to the jurisdiction of the Federal Communications Commission. Indeed, since the ARX tariff had been filed under this assumption, ITT Worldcom requested that the FCC expedite its approval of the ARX service. Its customers, explained the carrier, might seek alternative suppliers of automatic switching, suppliers not subject to FCC jurisdiction. Thus, on October 1, 1966, the Federal Communications Commission denied petitions for tariff suspension and ITT Worldcom's service took effect.[21]

As if matters were not complicated enough, RCA Communications, an international record carrier and a subsidiary of RCA, inaugurated a service called Aircon devoted to the needs of international air carriers. Aircon, a computerized message switching service, also contemplated the processing of seat reservations, inventory, accounting, data on cargo, freight information, as well as a wide variety of administrative services. RCA's service was thus dircetly competitive with that offered by ITT Worldcom. The difference between the two was crucial; RCA refused to file a tariff.

ITT Worldcom lost no time in urging the FCC to force RCA Communications to file an appropriate tariff schedule. As requested, the FCC asked for a statement of policy from RCA Communications. The carrier replied that its automatic information reservations service would not only be at a competitive disadvantage with those provided by Collins Radio, but filing a tariff would prejudice the issues posed in the FCC's computer inquiry which by this time had been announced. Thus, RCA Communications observed:

> It must maintain a competitive position that would enable it to compete effectively on a meaningful basis with non-regulated entities. Beyond this, a tariff filing would afford non-regulated companies an unfair advantage by exposing the entire Aircon service offering to their scrutiny while their offerings are beyond the regulatory purview.[22]

Where is the communication-data processing situation as it now stands? First, firms in both industries do not agree as to the precise limits and hence proper jurisdiction of data processing and data switching. Second, the domestic and international carriers are split among themselves as to what constitutes the relevant market for data switching; third, the international carriers are divied as to the FCC's jurisdiction over computer switching; and finally, the question of promotional pricing, and rate levels continue to beset carriers and non-carriers alike.

IV. Policy issues

A central theme can be detected in the confluence of events that prompted the FCC's investigation; and that is the question of establishing ground rules for market entry and competition. Consider this theme as it applies to three questions: 1) the leasing of communication lines, 2) carrier merger policy, and 3) joint cost allocation.

A. Market entry and communication lines

To repeat the obvious, regulated entities are attempting to engage in non-regulated activities, namely data processing, non-regulated firms are attempting to diversify into what is commonly regarded as regulated activities — communications. RCA Communications and Western Union typify the former trend; Bunker-Ramo typifies the latter trend. But as the Bunker-Ramo case suggests, diversification is not a reciprocal relationship. The carriers own and supply the nation's communication circuits, clearly an activity subject to state and federal regulation. With this as a base, it may be less difficult for regulated entities to diversify into unregulated activities than vice versa. In short, the granting of franchises is slow, expensive and hardly conducive to market entry for data processing firms.

What is important is that the carriers lease circuits to a growing group of firms who pose both as competitors as well as customers. This relationship can be expected to intensify; and as the Bunker-Ramo case testifies, market diversification poses no little problem to the Fderal Communications Commission. The Commission may find that carriers deny lines to companies who engage in communication on grounds that such activity is subject to regulation. On the other hand, the carriers may refuse to file tariffs on their data processing packages on ground of market competition. This asymmetrical relationship effectively bars market entry to major candidates who are seeking to offer computer utility services, particularly if those firms reside in the non-regulated sector of the economy. By foreclosing or conditioning market entry, the structure of this new industry obviously will be predetermined.

There is a deeper problem. RCA Communications' refusal to file a tariff on computer switching suggests a profound policy disagreement, at least within the ranks of the international carriers. At odds with both a domestic carrier (Western Union Telegraph Company) and an international carrier (ITT Worldcom), RCA Communications apparently assumes that its activities now and in the future will be competing directly with those services offered by non-regulated entities.

It is clear that the blurring of data processing and

communications does not lend itself to easy solution. In this search, the primary business test is an unsettling policy guide. Communication and computer firms will obviously employ the test to their own ends, however those ends are defined. That such a test borders on the arbitrary can be seen in its application. In the eyes of Western Union, flower and data processing service are "incidental" to communications. Yet the communication industry becomes less tolerant when the identical test is invoked by the data processing industry, to wit, the Bunker-Ramo case.

Thus, the primary business test cuts both ways; it can be employed to rationalize a movement from communication to data processing; or from data processing to communications. And if such a test is resurrected as a policy guide, it takes little imagination to visualize a backlog of adjudicatory proceedings in which the adversaries seek the magic number of 49%.

B. Market entry and merger

As noted above, non-regulated firms generally seek refuge in the primary business test as a means of avoiding "the burden of regulation and the obligation of service." But it is conceivable that a data processing firm may assume or perhaps even hope that its activities are a legitimate common carrier function. This may occur when a data processing firm merges with a regulated firm.

Any such consolidation poses two questions. First, does merger with a common carrier sanction the conversion of a non-regulated activity to a regulated activity. The answer is, of course, speculative. The merger, however, of the Ultronics Corporation with General Telephone and Electronic Corporation, the nation's second largest telephone carrier, suggests that this possibility is very real.[23] Indeed, it was Ultronics who argued that Bunker-Ramo's Telequote IV encroached upon a regulated communication activity.

Secondly, if movement from a non-regulated to a regulated status offers a solution to Ultronics, what options are open to Bunker-Ramo, Scantlin, and other rivals of Ultronics. Does the existence of direct competition among these firms stimulate what Kaysen and Turner term "parallel integration." And what are the implications of these activities in terms of the long-term environment for computer utility services. Merger policy, particularly the consolidation of non-regulated with regulated entities, raise provocative issues in terms of market entry.

C. Market entry and joint cost

Finally, the straddling of communications and the data processing package half regulated, half unregulated, conjures major accounting problems associated with joint costs. Like any regulatory body, the Commission is not entirely unacquainted with the problems of cost separation. Although it granted the telegraph company an accounting waiver for the data processing segment of the ARX service, the announcement by Western Union of its management information systems for Blue Cross and Dun & Bradstreet prompted the Commission to inquire as to the feasibility of separating the telegraph company's communication activities. In response, Western Union observed:

> Certain non-communication activities which are now clearly incidental to communication activities are expected to become significant in the future. The whole field of computer applications and information storage and retrieval is in a state of development; more important, many matters such as the method of operation remaining unsolved.[24]

Western Union's position of incipiency hardly eases a problem that is likely to become aggravated over time. The question of cost separation, in short, will undoubtedly command the attention of all participants in the FCC's computer inquiry.

That cost separation is no easy task is demonstrated by past experience. The carriers generally prove more receptive to discussions of rate of return rather than questions of rate structure. Even when the Commission persuades a carrier, as it did the Bell System, to calculate rate of return on fully allocated costs, the carrier tends to regard its studies as less than meaningful for purposes of public policies.[25] Thus, the search for cost separation standards within the regulatory sector is fraught with complexity. But more important, the search for standards applicable to a firm half in and half outside the bounds of regulation is equally imposing.

Yet, regulatory experience suggests that such separation is crucial. The telegraph investigation indicated, for example, that the Bell System was not above emplaying its monopoly markets to carry its competitive markets — all within the regulated sector.[26] No one doubts that the Telpak tariff has throttled the innovation of private microwave communication systems. If then, internal subsidization places a continual burde non regulation, the accounting hazards of a firm partly regulated and partly unregulated are no less challenging. To repeat, market structure and rate structure are inseparable. Both bear on the viability of market entry.

There is a final issue. The carriers are in one sense integrating forward, packaging communication lines

together with computer services. At the same time, the carriers lease circuits to potential competitors — the computer industry. Will wholesale, retail rates distort prices sufficiently in the final product market to rule out meaningful market competition? Bluntly stated, will the carriers accord themselves internal line discounts denied their customers? And if the carriers lease circuits from themselves or from other carriers, will such transactions, now apparently residing beyond the province of FCC jurisdiction, accord the carriers competitive advantages that ultimately spell who survives in the computer services industry?[27] Again, market entry rides on these issues, no matter how premature they might appear today.

V. CONCLUSION

In sum, the Commission's computer inquiry is confronted with an imposing agenda of policy questions. Not the least is the issue of definition and jurisdiction, for out of their determinations rests the question of market entry and market structure. Two options are clearly open to time-shared computer services: competition or regulation. But these choices apparently assume that the firm is restricted to the either/or extreme of data processing or message switching. The imperatives of technology suggest, however, that the pure case may be the exceptional case. As the amalgam of data processing and communications continues, a third choice, namely de-regulation, may pose as a policy alternative worthy of consideration. Perhaps it is unthinkable that a government agency cut the Gordian knot of regulation with any degree of seriousness. But the solutions to the computer utility question are sufficiently urgent to warrant broadening the search for unique policy prescriptions.

REFERENCES

1 Before the Federal Communications Commission. In the matter of regulatory and policy problems presented by the interdependence of computer and communication services and facilities. *Notice of Inquiry,* Docket No. 6979, November 25, 1966.

2 *Ibid.,* p. 1.

3 Bunker-Ramo letter to Federal Communications Commission, March 12, 1965.

4 Letter from the Western Union Telegraph Company to the Federal Communications Commission, December 3, 1965.

> As we understand the Telequote IV proposal, the service would be offered to the brokerage industry, the conclusion is inescapable therefore that under the proposal, there would be a holding out to the general public as that term is used in the law of common carriage.

See also American Telephone & Telegraph letter to the FCC, September 29, 1965, p. 2.

> It would appear that the transmission of communications is at the heart of its (Bunker-Ramo) proposal for 'message switching' services, which services might prove in fact a most significant element of Telequote IV. The so-called 'data processing' to the extent they exist at all in such cases might be deemed ancillary to the transmission of communications. In such circumstances, we believe a conclusion that Bunker-Ramo was not engaged in a common carried undertaking subject to regulation under the Communications Act would be open to serious question.

5 *Ibid.,* p. 3.

6 Bunker-Ramo letter to Honorable E. William Henry, Chairman of the Federal Communications Commission, August 23, 1965.

7 Letter from Ultronics Systems Corporation to the Federal Communications Commission, January 17, 1966.

8 Letter from IBM to Federal Communications Commission, February 15, 1966, p. 5.

9 *Ibid.,* p. 7.

10 *Ibid.,* p. 6.

11 *Western Union Annual Report* 1965, p. 10.

12 *Telecommunications Reports,* August 10, 1964, Volume 30, No. 37, p. 19.

> We are advised you do not intend to include in your tariff on file with the Commission charges for off-line processors furnished in connection with the system (ARS). It is requested that you advise us as to what you believe to be the legal justification for furnishing this item of equipment without having charges therefor on file in your tariff.

13 *Telecommunications Reports,* November 22, 1965, Volume 31 No. 51, p. 14. Western Union states:

> The type of computer equipment furnished varies according to the requirements of each customer. Depending upon the capabilities of the particular equipment furnished, such equipment may be utilized for electronic processing of business data and for electronic switching of message communications. The telegraph company also undertakes on request to provide programming of the computer equipment on an initial or a continuing basis.

14 *Telecommunications Reports,* November 29, 1965, Volume 31 No. 52, p. 3.

15 Letter from Western Union Telegraph Company to the Federal Communications Commission regarding computer lease and service arrangements, March 14, 1966, p. 10.

16 *Ibid.,* pp. 37-38.

17 *Ibid.,* p. 34.

18 In the matter of ITT World Communications Inc., Tariff No. 54, June 9, 1966.

19 *Ibid.;* Petitions of Western Union International for suspension; Reply of Collins Radio to reply of ITT World Communications, Inc. to petitions of the Western Union Telegraph Company, Western Union International Inc., and Collins Radio, Inc.

20 *Ibid.;* Reply of ITT World Communications, Inc. to petitions of the Western Union Telegraph Company, Western Union International, and Collins Radio, June 16, 1960, p. 5.

21 *Telecommunications Reports,* October 3, 1966, Volume 30, No. 43, p. 8.
22 *Telecommunications Reports,* February 27, 1967, Volume 33, No. 11, p. 6.
23 *Wall Street Journal,* May 16, 1967, p. 14.
24 Telecommunications Reports, September 12, 1966, Volume 33, No. 40, p. 16.
25 Before the Federal Communications Commission. In the matter of domestic telegraph service, Docket No. 14650, Report of the Common Carrier Bureau.
26 *Ibid.,* pp. 156-157.
27 Amendments to Title II of the Communications Act of 1934 (Common Carrier). Hearings before the Committee on Interstate and Foreign Commerce, House of Representatives, 88th Congress, Second Session, HR 6018, a bill to amend section 203 (A) of the Communications Act of 1934 as amended with respect to the filing of schedules of charges by connecting carriers; 1965.

Communications services— present and future

by W. B. QUIRK
American Telephone and Telegraph Company
New York

There is a fable about a king who was so vain that he wanted only his image on the coin of the realm. Therefore, he ordered his goldsmith to make him a coin with just one side. After many tries the goldsmith confessed that he couldn't make such a coin, and, in the best tradition of fairy tales, he had his head chopped off. We can learn a lesson from this story. Just like a coin, a computerized information system has two sides — communications on one side of the interface and a computer on the other. It isn't possible to construct a successful system by considering just one side. If one tries, he too may lose his head — or at least go out of his mind.

The telephone industry can speak with experience about total information systems because the nationwide dial telephone network is in reality a giant "computerized" information system. Even though the telephone network provides the user with communications services rather than data processing, the technology used closely parallels that of a general purpose computer system. Even the concept of shared use, which is the basic building block of the so-called computer utility, has been used for years in telephone switching systems.

When I try to convince people that there is nothing really new in the basic idea of a time-shared information system, I feel like Sisyphus, the legendary king of Corinth. He was condemned, you may remember, to roll a heavy rock up a steep hill in Hades, only to have it roll all the way down again when he got it near the top. Then he had to start all over.

I'm getting tired of pushing this rock up the hill, so for the moment I'll concede that perhaps the time-shared information system is something new. But I still feel sure that many of the lessons learned by the telephone industry over the years can be applied in planning and implementing computer utilities. Furthermore, we in the telephone business have an important role to play in providing the vital communications links which give life to computer utilities.

These communications channels can be compared to arteries or pipes. A pipe carries a stream of fluid; a communications channel carries a stream of electrical signals. A pipe can transport a variety of different liquids, so, too, for communications channels. Whether what goes in be voice, handwriting, machine data, still or motion pictures, or Beethoven's Fifth Symphony — so long as it can be translated into electrical signals — it can be transmitted.

The world's greatest arterial system for the flow of data signals already exists in this country — the nationwide telephone network. Today it reaches over 100 million telephone sets, and each of these locations can be arranged to send of receive data. The suitability of the dial telephone network to carry data signals was demonstrated by a series of Bell System tests in the late 1950's. The results of these tests were published in the now famous Alexander, Gryb, Nast Report.

Since then, of course, data communications needs have changed drastically — and they continue to change. The capabilities of the telephone industry are expanding right along with those needs, In fact, the real success of our business has been its ability to anticipate the future, to introduce new services even before users have requested them. For example, nationwide dial teletypewriter facilities were ready for time-shared computer use by the time prototype computers were being laboratory tested.

To date, Bell Laboratories has developed over 80 different models of data modems, 40 of which are now being offered to the using public. These together with the associated channels and a variety of station gear comprise a wide spectrum of data communications services. But we don't stop here. More are on the way, in a variety of common user and private line offerings — from narrowband telemetry services at one end to megabit and higher services at the other. I'd like to tell you about some of these new services, paying special attention to communications costs and interdisciplinary problems.

We talk about *service* rather than about *hardware* because service is what we offer — our end product. Hardware is only a means to an end. The two most important ingredients of the services we offer are innovation and reliability. I want to direct my comments today to the first one — innovation — and skip the second. But I want to assure you that the telephone industry doesn't skip reliability; reliable service demands hard work on the part of many people.

We think it's important to provide service rather than hardware because it allows the greatest scope for flexibility, economy and innovation. Improvements in technology can be introduced when and where they will provide the greatest benefits to users. The careful phasing in of technological innovations is essential in keeping facilities cost down. Naturally, this has a beneficial effect on rates, even though cost of equipment is only one factor in setting rates. Similarly, by avoiding unnecessary duplication of useful plant, we also help keep costs down.

There are computer users who want the telephone industry to serve them only with the new digital transmission systems, instead of leaving the choice of facilities to the telephone company. They claim that this would let them get more data thruput for their dollar. Such proposals are impractical. They fail to recognize that perfectly usable plant would be unnecessarily duplicated by such digital systems. To do so would mean higher cost to the telephone companies. What's more, they ignore the prohibitively high amounts of capital needed for such an undertaking.

Others suggest that the best way to lower the cost of data communications is to establish a totally new digital network provided by a new data communications common carrier. This would duplicate the high-quality data services offered by present common carriers. To provide the using public with the same data service capabilities as available with DATA-PHONE service would mean a digital network the size of the present telephone network!

The advantages of planned, systematic innovations are demonstrated by the Bell System's T-Carrier program. We are now installing T-1 digital carrier systems to meet growing communications needs — both voice and data. These systems, by using pulse code modulation, can transmit up to 1.5 megabits per second or 24 simultaneous voice calls over two regular twisted cable pairs. We are adding these facilities wherever the needs warrant it. Today these digital systems are available only in selected locations, but eventually a large part of our plant will be of this type. Even higher speed digital systems are now being developed. A 281 megabit per second T-type facility is planned for 1971.

Our program for introducing T-Carrier is somewhat analogous to the way airlines have phased in jet aircraft. They didn't immediately junk all their propeller-driven aircraft. Instead, they added jets where they could do the most good, to improve service and to reduce costs. We have the same objectives for T-Carrier.

Innovations are also being planned to improve the effectiveness of our existing plant to carry data signals. For example, a new DATA-PHONE data set, capable of transmitting 3,600 bits per second over the dial telephone network, is now in the technical trial stage and will be available in the near future. A similar data set has been developed for private line service, but this one is able to handle 7,200 bits per second. Even higher speed versions are being planned for later on.

We are constantly looking for new ways to meet the expanding data communications needs of the public, including the need for lower priced new services. One such approach, now being studied, is a limited distance data communications system which uses extremely low-cost data sets and line concentrators. Such a system looks especially attractive for use with a time-shared computer at a university campus, in an industrial complex or within a community or other limited geographical area. As planned, this system would be able to serve distant terminals at a somewhat higher cost, but the greatest cost savings would be for stations within a limited service area.

Data line concentrators and multi-channel data sets (multiplexors) are being developed. The sub-channels of these data sets will match the bandwidths of our other narrowband service offerings. For some time now, our tariffs have allowed customers to do this kind of channel deriving on our private line voice channels, using their own equipment.

At the wide band of the data spectrum we find requirements for the transmission of occasional high speed bursts of data — to update computer memories, to load-share a computer or just for back-up in case of computer failure. While these and other applications need wideband channel capacity, they can't justify the costs of full-time wideband private lines. Recognizing this need, we are now introducing on a trial basis a service called DATA-PHONE 50 — a 50 kilobit per second common user switched service. Initially, it's being offered in Chicago. Additional trial offerings may be made in other cities in the near future.

These and other planned new services reflect a continuing research and development effort that began way back in the earliest days of the telephone. It

continues today in the realms of both voice and data communications. Our basic function hasn't changed at all; the computer and its requirements have simply added a new dimension to our historic function.

Communications common carriers **are** the vital link between the computer and those who use it. We provide the transparent channel to interchange signals. We provide some input-output devices, such as teletypewriters and TOUCH-TONE telephones. And we are working constantly with designers of computer systems to establish plans for total computer communications systems. In this process we are blending old technology with new—Rembrandt with Picasso, Bach with the Beatles. If we can be as creative as they are—or were, computers and communications will make beautiful music together for decades to come.

Communication needs of remotely accessed computer

by W. E. SIMONSON
University of Southern California
Los Angeles, California

There are a variety of problems posed by increasing interest in remote access to computers. These problems are further compounded by the circumstances facing firms such as mine which have the audacity to attempt a commercial offering in the face of these problems. This entire area has been much the center of considerable attention as a result of the FCC notice of inquery. The question has now been raised as to what are the real demands of a remotely accessed computer complex.

On the broadest and most philosophical level there is a problem in the basic environment of a highly regulated utility, especially in the face of a rapidly changing technology. The basic proposition that it is possible for a group of men without substnatial exposures to intricacies of this business to adequately regulate communication is suspect. This is not intended as criticism of the job done by the commissioner, but a question of the assignment. It is my firm belief that except in the case of natural monopolies (montonic decreasing cost functions or industries where the public inconvenience and implied cost of competitive service would be disruptive) regulation should be aimed at creating and maintaining a comptitive environment. Any attempt to engage in regulation of substantive issues (specific rates, services) presumes a competence on the part of regulatory bodies which is hard to maintain. If such regulation to substantive issues were felt necessary in the computer industry, I feel that it would clearly require a specialized agency. Can you imagine the problems of setting tariff schedules on custom programming or on such things as a payroll service considering all the variations that such a service could provide.

Let me develop my thesis in the context of the services of a commercial time sharing venture. What are the communication services require by the firm and its customer? They include terminals, modems, and lines. Of these, only the lines or channel capacity are inherently a common carrier function (even though a common carrier might well do a better job of providing the others). Even here it is possible to consider private microwave.

Private microwave offers a good case to illustrate my point. There is a basic economic consideration as to whether or not it is economic for a user to consider. The competitiveness of microwave can be reduced beyond economic arguments by having arbitrary restrictions on interfacing microwave lines with the lines of a common carrier. Where interface specifications are met, say by using the common carriers' equipment, in my opinion no objection should be raised.

In another example the ownership of a piece of communication equipment obviously does not affect its performance and hence should not be an issue. If one uses AT&T lines there is no clear reason why Bell Modems need be used. In the case of private lines this is no longer true. Restrictions do apply to the dial-up network. The need for some control of the signals to go through the switched network cannot be argued. However, if GTE modems give acceptable signals coming from an independent telephone company, they should also yield acceptable signals from a private firm. Similarly, other models that conform to interface specifications should be acceptable.

This does not mean that I would suggest that one is better off with non-Bell equipment; quite the contrary. However, if the prices Bell charged got too far out of line or their service too inferior they would than run the disk of losing business. This competitive pressure will do for more to ensure that Bell offers the best service they are able to than any amount of FCC inquery. Similarly, individual users might chose to accept lower levels of safety as far as reliability is concerned in order to get faster transmission. If AT&T feels that they cannot announce service, they are not prepared to make generally available, then the customer should have the right to run the risk of using faster devices within a Bell System guarantee.

The entire question of technological development is a serious one. The more closely regulated an industry is the slower technological progress will tend to be. The administrative process alone will tend to slow down the provision of new offerings. Similarly procedures used in rate base calculations can serve to remove economic incentive from new developments. Since the development of new transmission technologies, the savings they can bring have substnatial affects on the economics of remote computation. It is of particaular interest to our industry that means be found to encourage the offering of new services that represent the best state of the art techniques. If there are risks in the utilization of such services or if it is not practical to make them uniformly available, restricted offerings should still be available. In light of recent FCC decisions on the regulation of AT&T's rate of return, an examination of incentives to provide new state of the art offerings may have increased importance.

Let us turn to the areas of message switching. A question has been raised as to whether message switching is inherently a common carrier function. Although the channel capacity is purchased from a common carrier, we fail to see how the public is benefited by restrictions on the nature of the data sent down the line. The situation is analogous to having the computer manufacturers say they will happily sell you a computer but tell you that you are not allowed to run your payrolls on the machine. For that purpose you must buy time on their data center machine even if you have substantial quantities of idle time on your own computer.

One of the problems in the regulation of major portions of this area is that the common carrier himself constitutes a court of first resort. If he thinks you propose a use of his lines which he feels is improper he will refuse to sell you the lines. Now you carry the burden of proof and its attendant legal costs. Minimally, it would appear that it would be only fair to presume that the service is leigtimate and force the common carrier to prove that he should be allowed to discontinue service.

This situation is exemplified in the Bunker Ramo case where Western Union had refused to sell lines for a version of Telequote (because Western Union felt that features of the service impinged on common carrier prerogatives). Bunker Ramo then negotiated a system specification that AT&T felt acceptable. Though this explication is oversimplified, it bothers us that the common carrier, which in the case of Western Union desires to be a major competitor in the service bureau industry, should have the prerogative of deciding whether or not a service violates the law. We feel that they should be required to sell to anyone and, if they feel that a particular use is not legitimate, they should bear the burden of proof as to why they should be allowed to discontinue service. This important difference would give the brunt of the legal cost and risk to the common carrier and would give the user service during the period of dispute.

It is clearly a matter of opinion in many cases as to whether a specific computer service provides message switching functions. Not only commercial services are affected, but those of any firm providing service or communicating with its customers and suppliers. In an area this unclear it would appear to me that (1) there is no public interest served by a prohibition of message switching; (2) if prohibited, any decisions as to the presence or absence of a violation should be made only by a disinterested party.

The last item that I would like to suggest is the desirability of including line standards and transmission characteristics (envelop delays, noise, etc.) in the tariff filings (perhaps with deviations by city or major trunk). This might be helpful in having a standard against which one could test in case of disputes on the quality of lines. As I am sure many of you are aware, problems arise when multiple suppliers are involved in an installation. Hardware suppliers typically blame lines for most problems and vice versa. Though we have always found Bell to be cooperative in trying to run down the source of problems, an objective standard against which one could test could aid in the resolution of many of these disputes.

In closing, then, let me summarize by simply verbalizing the hope that these ideas will at least prove sufficiently provocative to stimulate some thought and new ideas. If so, they will have served their purpose. If nothing else, the FCC inquiry should demonstrate the need for considerable thought. Ignoring the specific illustrations, I have tried to advance the thesis that the needs of remote computational usage would be best advanced by as open, competitive, and generally unrestricted an atmosphere as can be created. Where regulation or restriction is deemed absolutely essential, every effort should be made to see that such regulation does not retard technological progress.

Another look at data

by GEORGE H. MEALY
Computer Consultant
Scituate, Massachusetts

INTRODUCTION

We do not, it seems, have a very clear and commonly agreed upon set of notions about data—either what they are, how they should be fed and cared for, or their relation to the design of programming languages and operating systems. This paper sketches a theory of data which may serve to clarify these questions. It is based on a number of old ideas and may, as a result, seem obvious. Be that as it may, some of these old ideas are not common currency in our field, either separately or in combination; it is hoped that rehashing them in a somewhat new form may prove to be at least suggestive.

To begin on a philosophical plane, let us note that we usually behave as if there were three realms of interest in data processing: the real world itself, ideas about it existing in the minds of men, and symbols on paper or some other storage medium. The latter realms are, in some sense, held to be models of the former. Thus, we might say that data are fragments of a theory of the real world, and data processing juggles representations of these fragments of theory. No one ever saw or pointed at the integer we call "five"—it is theoretical—but we have all seen various representations of it, such as:

V $(101)_2$ $(5)_8$ 5 0.5E01

and we recognize them as all denoting the same thing, with perhaps different flavors.

We could easily resurrect disputes in medieval philosophy at this point! The issue is ontology, or the question of what exists. While a Platonist would claim that even universal concepts, such as redness, exist independently of whether anyone perceived them properly or at all, a conceptualist might claim that we perceive only ideas and they have no existence until they are perceived. No doubt, both of these gentlemen would quarrel with what I have already said. My friend the nominalist, however, would permit me to entertain such notions, so long as I did not insist upon his treating them as anything but words.* Since it happens that the following does not depend on any particular ontology, we can avoid a quarrel by adopting the nominalist's position.

Our plan of attack is to indicate the nature of the theory of relations, based on the example of genealogical data. This will lead immediately to formulation of our notions about data in general, including rather precise definitions of concepts such as data structure, list processing, and representation. These notions are used in the second part of the paper as the basis for some remarks and suggestions concerning language and system design.

Toward a theory of data

Relations

To fix our ideas, consider the following example of genealogical data, taken from Reference 2:

... SNOW ...

4. HENRY (7) [Henry (6), David (5), Anthony (4), John (3-2), Nicholas (1)], b. 18 Sept., 1810; m. 13 Dec., 1840, Susan Stoddard, dau. of John and Betsey (Stoddard) Lincoln. She was b. 21 Aug., 1822, and d. 13 Sept., 1880. He d. 25 April, 1904. "Master mariner." Resided in house which he built on So. Main St. south of his father's.

Ch., b. in Coh.,—

i. SON, 17 April, 1841; d. 6 May, 1841.
ii. JAMES H., 28 June, 1842.
iii. ANN FRANCES, 24 Aug., 1844; d. 5 July, 1869, unm.
iv. SUSAN ELIZABETH, 20 Oct., 1846; m. 1 Jan., 1869, Leonard A. Giles, Troy, N.Y., She d. 25 April, 1827.
v. RUTH NICHOLS, 29 June, 1848; m. 24 Jan., 1892, James H. Nichols.
vi. CHARLOTTE OTIS, 8 Nov., 1850; m. 5 Mar., 1879, George W. Mealy, Troy, N.Y.

*See W. V. Quine's essay "On What There Is" in Reference 1 for an interesting and frequently entertaining discussion of these points of view.

vii. BENJAMIN LINCOLN, 2 Aug., 1853; d. 24 Jan., 1859.

The above, of course, does not record more than a few facts concerning Henry(7) Snow, Jr.; we are already in the realm of symbols. The nominalist is equally prepared to be told that James H. Snow is a misprint or that he is alive today. Officially, the nominalist is under no illusions about data: A data base never records all of the facts about a group of entities; a fact may be recorded with complete or lesser accuracy; and non-facts may be recorded with equal facility.

It was, no doubt, the study of genealogical data that led to the invention of the theory of relations, which will lead in turn to our notions about data in general. Informally, relations are simply a generalization of family relationships, and genealogical data is one of the older instances of recorded data.

We start with a set of individuals (or any other type of entity) and a second set, which may or may not be the same set as the first. A *relation* is a correspondence between members of the two sets. For instance, son-of, children-of, father-of, ancestor-of, sib-of, and the like are all relations, as are birth-date-of, occupation-of, age-of, residence-of, marital-status-of, etc. The children-of relation, in the case of the Snows, is a correspondence between Henry (7) and his children, Henry (6) and his, and so forth. To be somewhat more accurate, there are at least two possible children-of relations; due to the possibility of remarriage, for instance, the set of children of a given pair of spouses is not necessarily the same as the set of children of a given individual of that pair. This is to say that, in general, relations are correspondences between n-tuples from a set and m-tuples from some possibly different set.

Another manner of speaking about the same subject material is also in common use; we speak of some set of things, *attributes* of those things, and *values of attributes*. Attributes are the same as relations, being a correspondence between the things and the values (which may also be things). Still another manner of speaking is to use the term "*map*," as we shall do shortly. A final manner of speaking is to use the term "*function*"; in the theory of relations, this term is reserved for the special case of relations which are single-valued.

The notion of attribute should be distinguished (as PL/I does not) from that of *property*. To say that something has a given property is to say that some attribute of that something has a certain value. Thus, when I say that a house is red, I mean that the value of its color attribute is red, not that I intend to identify the house with the universal concept of redness. Properties may be combined using the usual logical connectives to form new properties, unlike values. Thus, the tall, red house has a property not shared by the long, red house, except by accident. Its color attribute has the value red *and* its height attribute has the value tall. PL/I would with more precision state that the variable X has the property FIXED BINARY DATA-TYPE; the data type is the attribute, and FIXED BINARY is its value, or part thereof, as we shall see.

Before proceeding, it is worth noting that the genealogical example illustrates two types of relations which seem to be qualitatively different. On the one hand, we have the family relationships and, on the other hand, we have relations between individuals and elements of other sets, such as dates. There are other relations which are more akin to family relationships, such as the relations between people who live in the same dwelling or who work for the same organization. An organization chart also illustrates this type of relation, which we will call "*structural.*"

Data maps

We have called data fragments of a theory of the real world. It is now time to examine the nature of that theory. Along the way, it will be possible to propose precise definitions for many terms which are normally used rather more loosely. We start with the (undefined) notion of a set and introduce the notion of a map more precisely than we did above. We will use standard notation from set theory: Sets will be represented by capital, Roman letters and elements of sets by lower case letters. If a is an element of the set A, we write

$$a \in A \cdot$$

If A is a subset of B (that is, all elements of A are also in B), we write

$$A \subseteq B \cdot$$

Maps will be represented by Greek letters. In talking about a map of the set A into the set B (that is, the map makes values in B correspond to arguments in A), we will write

$$\mu: A \rightarrow B$$

or

$$A \xrightarrow{\mu} B \cdot$$

The latter form is useful in diagrams displaying several sets and maps.

We require of a map that it assign to each element of A either nothing or one or more elements of B; some members of B may not be assigned to any element of A. If every element of B is assigned to at least one element of A, we call the map "*onto.*" If a unique element of B is assigned to each element of A,

the map is called "*one-one.*" A one-one onto map pairs each element of A with a unique element of B and *vice versa.*

We will write ordered n-tuples of set elements as a parenthesized list:

$$(a_1,a_2 \ldots ,a_n)$$

In such an n-tuple, if each a_i comes from a corresponding set A_i, then the set of all possible such n-tuples is written as:

$$A_1 \times A_2 \times \ldots \times A_n$$

If all of the sets A_i are the same set A, then the set of all n-tuples from A is written as:

$$A^n$$

The set of all subsets of a set A is usually symbolized as

$$2^A$$

owing to the fact that there are exactly 2^n possible subsets of a set of n elements. Figure 1 displays a two element set A together with the set of pairs of elements in A and the set of subsets of A.

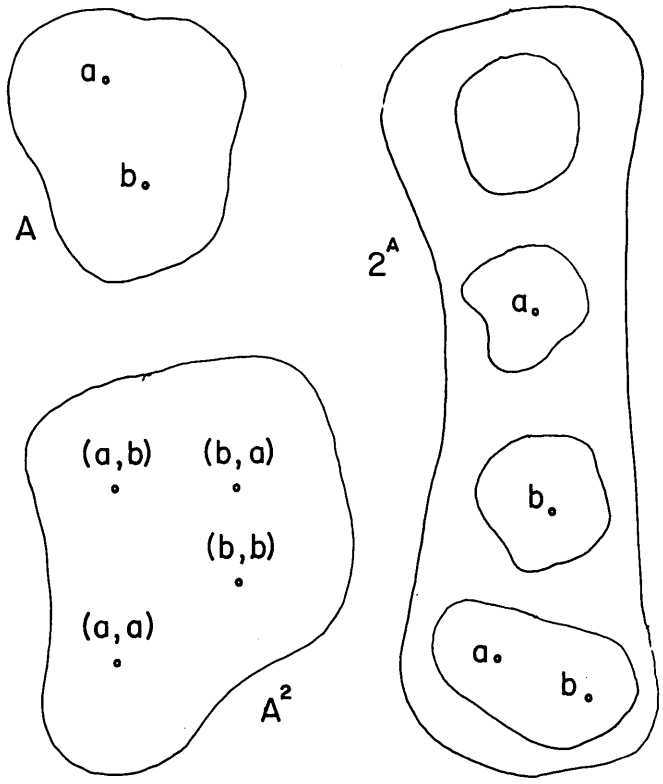


Figure 1 – Sample sets

Now data are supposed to record a set of facts about some set of entities, be they real or abstract. In our present formal manner of speaking, we can contemplate a set of *entities*, E, a set of *values*, V, and a set D whose members are maps of the form:

$$\vartheta : E \rightarrow V$$

We will call D a set of "*data maps*". As we shall see later, D is a set whose elements are subsets of the set $E \times V$. That is,

$$D \subseteq 2^{(E \times V)}.$$

In the case of the Snows, one of the data maps in D assigns to Henry(7) his birth date in the value set of dates. The same data map assigns a birth date to James H. Snow. The date-of-death data map assigns no death date to James (at least, not on the basis of the evidence quoted earlier); its value for him is *undefined.* The father-of map assigns Henry (6) to Henry (7), but its value for James H. Nichols is again undefined. In the case of this map, the value set V must contain E as a subset; we have an instance of a *structural map,* which we define as any map of the form:

$$\sigma : E^n \rightarrow E^m$$

for some non-negative n and m. This is, by our previous definition, a data map only if $n = 1$.

The set of entities, E, might be larger than one might expect. For technical reasons, it is often convenient to introduce certain auxiliary entities. For instance, in the genealogical example, it might be appropriate to introduce entities corresponding to married couples as well as the entities representing individuals. Again, one normally wishes to record information that has no direct bearing on the entities the data are about, such a file name, retention date, etc. In this case, we merely augment the set E with a special element e_0 together with data maps defined only for e_0.

We have not inquired into the possible structure of the value set V, nor have we admitted all possible structural maps as candidates for the set of all possible data maps. The motivation for this is to simplify matters by considering only the structural data maps

$$\sigma : E \rightarrow E$$

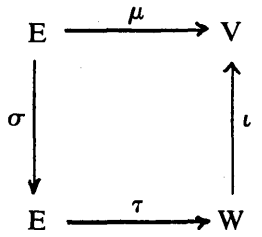
and by restricting the data maps to be functions – that is, to have a single value for each argument. By appropriate adjunction of additional elements to E and data maps to D, this can always be done.

V, itself, might be considered to be constructed from other sets. For instance, in the definition of data maps we have implicitly assumed that E is a subset of V in order to admit structural data maps. The set composed of the elements of V which are *not* elements of E, that is

$$W = V - E,$$

could have structure, however. It might be a set of ordered triples or something more complicated, such as a set of vectors whose elements are vectors, and so forth – in other words, a set of tree structures. *On the contrary,* we will insist upon explaining structure by using structural data maps, and it will then be the case that all data maps can be defined in terms

of a non-structural map τ applied to a structural map σ. That is, if μ is an arbitrary data map, the situation shown in the following diagram can be made to obtain.

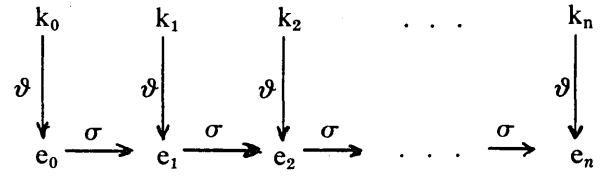


In the diagram, ι is the identity map of W into V, expressing merely that W is a subset of V. The diagram expresses the fact that, starting with an element of E, one gets the same value by applying the map μ as by applying σ, τ, and ι in that order.

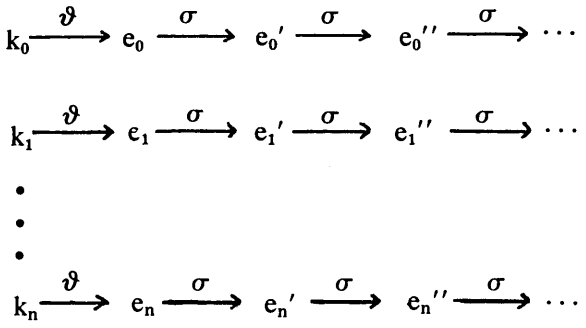
Access functions

We need a way of travelling around over the entities, or locating them either relative to each other or from something roughly like names or addresses. This mechanism is provided by the notion of an *access function*, which will be any map whose value set is the set of entities or a subset thereof. Access functions are not necessarily structural data maps, as we shall see immediately.

Certain special types of access functions fall out at once. Suppose we have a set K (for "key") and a map from K which is one-one into E; this is a *direct access function*, for we may regard K as a set of keys, addresses, or names of entities in E. On the other hand, suppose that we have a structural data map and some initial element of E, say e_0; if each application of the map gives us a new element of E, until the final application gives us nothing, then we say that the map is a *sequential access function.* It mimics the behavior of the successor function of elementary number theory, except that E is finite rather than countably infinite. These two situations can be illustrated by the diagram:



Note that we have not required that the maps be onto —a map into may or may not be onto. A case in which the maps are not onto is given in the following diagram:



This is reminiscent of the indexed sequential access method of Operating System/360.

Data

We have still not explained which of the objects above we mean to regard as data. The data maps themselves, or rather their elements, will be so considered, but this choice requires further justification.

What happens when data are processed? Our naive notion is that values are used as arguments to the procedure and the result is that values get defined, redefined, or undefined, so that the value set V must be regarded as being the data. But, what if V is the set of integers? It certainly cannot be the case that data processing redefines the integers! Nor are the entities changed, so we are left with the data maps. This notion may be a bit hard to stomach at first, but may be made more palatable by considering ways in which a map may be specified:

- We may specify a test which decides whether or not the map actually assigns a given value to a given argument.
- We may have a rule, or procedure (such as applying several maps in succession) which will designate the value, given the argument.
- We may define the map by actually exhibiting the pairs of corresponding arguments and values.

In the theory of relations, the third approach is usually used to define a relation—it is simply a set of ordered pairs.

In our case, suppose that

$$\mu(e) = v$$

This is the same as saying that the ordered pair (e,v) is a member of the set μ. To redefine the value of the *data item* (e,v) is to redefine μ by removing that pair from it and adding a new pair (e,v'). This justifies our earlier statement that

$$D \subseteq 2^{E \times V)}$$

Thus, data processing changes neither E nor V, as desired.

We have, incidentally, slipped in a definition of data item, which is an element of a data map. A *data element* will be the set of all data items associated with a given entity. List elements in IPL-V and LISP are data elements, in this sense. The notion of a logical record also corresponds to data element in our sense, and field corresponds roughly to our data item.

This explanation of data processing may seem quite artificial, in view of our Platonistic feeling that the "right" rule for assigning the value of a data item should be independent of how we do our data processing. My friend the nominalist would not be bothered by this scruple—he did not claim that such a thing as a "right" rule existed in the first place; data do not necessarily represent facts with utter accuracy. Data processing, he might say, is data's way of attempting to adjust to the facts, if such there be.

Procedures

We have now noted the effect of a procedure—it redefines one or more data maps or, what is the same thing, changes the value part of certain data items. The effect on D is to map it into a new subset of the data maps. In other words, *procedures* are maps of the form

$$\pi : 2^{(E\times V)} \rightarrow 2^{(E\times V)}.$$

Our idea about D is that it is the data at any given moment of time, not the data for all time.

The import of our introduction of the auxiliary entities was to effect a clean separation of structural from other considerations. That is, we have set things up so that any data map can be decomposed into a structural data map followed by a non-structural data map. The structural data maps are maps of E into E, by definition. Our long-standing name for data items in such maps is "*pointers.*" This, in turn, suggests an identification of *list processing* with procedures which process structural data. A list processing procedure, hence, is any map of the form

$$\lambda : \quad 2^{E^2} \rightarrow 2^{E^2}$$

This is a precise version of our vague notion that list processing has something to do with pointers and data structures.

Data storage and representation

The foregoing model obviously can be taken to apply directly to physical storage media.[3] To entities correspond cells in storage (blocks, words, characters, bits, registers, etc.). Maps specify attributes of the storage cells (more properly, properties) such as content, structure, parity, ability to read and/or write, address, protection key, and the like. The structural maps and access functions clearly correspond to our more usual notions of storage structure and access.

If our data maps are an abstract theory of the real world, we must do data processing with something else; computers are, after all, not abstract objects. However, the abstract theory is just as capable of modeling computation as it is of providing models of the real world—possibly even more so. We are confronted, we might say, with three *systems* in any specific situation. Each such system is composed of a quadruple of entities, values, data maps, and procedures. The first system is, at least from a Platonistic point of view, some part of the real world, the second is our theory of the first, and the third is a machine representation of that theory. A *representation* is, itself, now defined as a map establishing a correspondence between two systems.

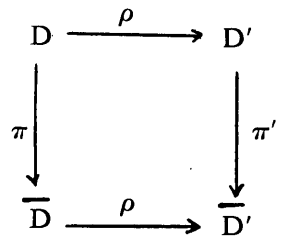
What criteria should a representation satisfy? Well, consider a system in the above sense:

$$S=(E,V,D,P)$$

where P is the set of procedures. Further, let π be any procedure in P, mapping D into a new set of data maps D, and let ρ be a representation map which maps S into S':

$$\rho: \quad (E,V,D,P)\rightarrow(E',V',D',P')$$

For any object in S, we wish the representation to assign a unique object in S', and *vice versa.* In other words, ρ should be one-one onto. However, we desire more than just this; in order to insure that anything happening in the one system also happens in the other, we require that the following diagram be *commutative:*



or, in other words, that:

$$\rho\pi = \pi'\rho$$

This criterion can fail in two ways: (1) obviously, when the map ρ is not one-one onto, and (2) when the procedure π', chosen in the belief that it corresponds to π, does not in fact so correspond. It might be thought that the second alternative can happen only by mistake, since we could presumably define the

procedure maps in terms of E and V and they are mapped into their primed counterparts in a one-one onto fashion. The rub is that, in practice, the first condition frequently does not obtain, and this gives rise to doubt as to which procedure π' *best* represents an abstract computation π.

The reason ρ fails to be one-one onto, usually, is that the set V′ is of a different size than the set V. The most obvious example is the case of machine representation of the real numbers; only rational numbers may be represented with complete accuracy on a machine (numerically, at least), and our common floating-scale representations are capable of representing only a finite number of rationals at that.* Moreover, the primitive operations out of which we compound procedures are only "best" representations of the abstract operations we have in mind. The size of the literature on floating-scale representations and arithmetic testifies to the amount of disagreement existing on what "best" means!

In the case of a machine representation, we have an abstract system and a representation map mapping it into the machine system. In fact, we frequently employ more than one machine representation for certain data items (e.g., numeric data). On the other hand, there is the physical storage system, and the machine representation of the data and procedure maps must be mapped into the physical storage. The structural part of this mapping—that is, the correspondence between the structure of the data and the structure of storage, we call the *data organization.* While this *enables* data access, it is not access. Access is a feature of the processing of the data, not of the data itself or how it is represented; different procedures will, in general, want to access the same data in different ways and orders. The order in which data items are fetched and stored is (or should be) independent of the data organization; this notion was an important principle in the design of the data management subsystem of Operating System/360.

Data description

What do we mean in general by the term "data description?" We might be tempted to confine our attention to the theoretical realm and, like ALGOL, talk of real numbers, integers, dynamic own arrays, and the like, considering it the job of any representation to be faithful to our theory. To do so, however, would be to beg the question by ignoring descriptive information which we use every day, and in machine-processable form at that. For example, the COBOL Data Division contains information which we would call data description, such as field lengths, and these certainly describe the machine representation, albeit incompletely. Information describing a data set appears in the volume and data set labels, and still other information is used to compile the procedures. All of this information, and more, is data description, although not all of it is stored explicitly or even in one place, nor is it available at all times. Some of it exists only in the minds of men!

So, I will be dogmatic: Data description describes *machine* data systems, representations, and organizations, rather than abstract data itself. That is, it is a specification of the maps, usually in terms of procedures which will accomplish the mapping, and the salient characteristics of the entity and value sets. To describe a data aggregate—that is, a file or data set—we supply this information together with information concerning the aggregate as a whole.

The term "data type" has been used informally above, but not in any essential manner. Intuitively, we feel that the data type tells us what kind of data we are dealing with. Had we not discovered that data items must be regarded as being elements of data maps rather than elements of value sets, our first false start toward a definition of data type might be to regard it as an attribute of the value set. A moment's reflection, however, is enough to convince one that this is an untenable notion; a given bit pattern may represent an element of any one of a number of value sets (binary integer, floating hexadecimal number, character string, or something else). In fact, it is not unusual to treat the content of a given storage cell as if it were of one data type at one point in a program and of another data type somewhere else.

Our next theory about data type, then, might be that it is related to the kinds of procedure maps used to process the data items. We retreat to the abstract realm, and consider the data type as specifying the mathematical system which governs processing. We might then come out with the following generic data types:

- *String* —free monoid on a finite number of tokens
- *Boolean* —Boolean algebra (or, equivalently, ring)
- *Numeric* —field (or more general system)
- *Pointer* —directed linear graph

This doesn't help very much. We distinguish between fixed binary and double precision complex floating decimal data, and this false start manages to keep us from making distinctions of practical importance by preventing us from talking about representation.

*Matula[4] has studied the maps which convert a floating-scale number from one radix to another. He shows that the map may be one-one or onto, but not both!

Nevertheless, the above are not irrelevant to data type; they are merely incomplete definitions. We are forced to conclude that a data type is a fragment of data description and, as such, describes a portion of a system and its associated representation and organization maps. It is an attribute of entities. We can tie the notion down better by looking at further determinants of data type for each of the generic data types mentioned above:

String

- Code—the code for each token (character, bit, etc.). E.g., USASCII, EBCDIC and the like.
- E.g., USASCII, EBCDIC and the like.
- String length—fixed or variable, and value of length attribute.
- Justification—left or right, and padding token.

Boolean:

- Code—the code for each truth value (or n-tuple).
- Field length.

Pointer:

- Code—machine address, base and displacement, item number in table, etc.
- Code for null pointer.

Numeric:

- Code—digit code, radix or weights, excess.
- Sign treatment—unsigned, sign and magnitude, radix or radix minus one complement.
- Scale—fixed or floating, value of scale.
- Rounded or truncated calculation.
- Arithmetic algorithms.
- Numeric limit(s) on value.
- Field length, or precision.
- Aligned or packed (storage mapping restriction).

It is evident that a complete description of a given data type contains information concerning representation of elements of the value set; information stating which representation, procedure, and organization maps are applicable; and data to be used in any mapping (such as value of scale). It is further evident that two entities which are not identical in all of these respects should not be regarded as having the same data type, unless the variable information is stored as data to be used interpretatively in accomplishing any of the mappings. Thus, variables of type real in ALGOL must be regarded as having different data types as represented, say, on the IBM System/360 and the CDC 6600.

Thus far, we have not discussed structural data. We even seem to have done a bit of violence to the notion of separation of structural from other data, but this is easily fixed. For instance, consider a floating-scale entity. Both the mantissa and the characteristic must be available during calculation, and they are treated as one entity during floating-scale arithmetic but as two entities during radix conversion. To be completely consistent, for each floating-scale entity we should introduce two more auxiliary entities; each of these is a fixed scale number and can be described appropriately. Similarly, we can take care of complex and multiple precision entities.

Turning to purely structural data, it is clear that we can invent data types, such as arrays, lists, tables, ring structures, etc., for any sort of structure worth classifying. In practice, of course, we suppress most of the detail involving the auxiliary entities and write down descriptions like:

```
DECLARE A(3,3) FIXED BINARY (15);
```

to describe a three-by-three array of 15 bit fixed-scale binary integers.

Languages and systems

On the basis of the point of view about data advocated earlier, more light can be shed on several issues of current interest. In some sense, these issues are all related to the possibilities of flexibility in choice of machine (machine independence), choice of data representation (ability to define new data types), or ability to strike a balance between compilation and interpretation (variable binding time).

Representation independence

For some years now, one of the more persuasive arguments in the favor of narrative languages such as ALGOL, COBOL, FORTRAN, JOVIAL, NELLIAC, and PL/I has been that they have offered some measure of machine independence. That is, the user is offered a greater or lesser degree of hope that a program written in a particular language together with data processed by the program can be processed on a variety of computing systems, with something like the same results.

Independent of one's personal degree of confidence in fulfillment of such an objective (and most of us believe that the objective is a Good Thing), I would urge adoption of the term "representation independence" as being more appropriate than the term "machine independence." In espousing such an objective, one's philosophical point of view might be that data processing takes place in the abstract realm in which our theory of data is formulated. It is, following this line of thought, more or less an accident (economic issues and questions of accuracy aside)

which computing system and machine representations are chosen in order to do the actual processing, and all such choices should lead to the same results.* In principle, I cannot quarrel with such a view; in practice, I wonder if blind pursuit of the objective does not often result in prejudgment of the economic issue. I hasten to add, however, that most programmers tend to be *too* pragmatic—in the long run, generality often costs less than specificity.

Representation independence is a more stringent objective than is machine independence. Representations are equivalent only when the representation maps are one-one onto and commute with the procedures, as we have seen, and there are cases in which this simply cannot be achieved in practice. Even in cases where one can choose a representation which satisfies our criterion, the cost may be unacceptable and we are forced to make do with a "best" representation. This should be counted as no disaster; it simply means that in practice we must relax the criteria of representation independence sufficiently to stay within optimal bounds of accuracy and economic data processing. Paradoxically, the best way to follow the spirit of our objective is to recognize that we can't live up to the law in all cases.

Our Platonistic tendencies have led us, in the past, to attempt to banish considerations of representation from language design and usage. I would suggest, contrariwise, that any serious attempt to design languages which are significantly more representation independent that at present is doomed to failure unless the notion of representation is made part of the language, ungrudgingly. At that point, we are again faced with the issue of how cleanly we can separate specification of the algorithm from specification of the data representation. Possibly, this introduces a new version of the UNCOL controversy! To overstate the case, the notion behind UNCOL was that one could pump a language description into one end of a compiler, a machine description into the other end, and come out with quality compilations. While many would still like to believe this, the evidence is that the two descriptions cannot be separated that cleanly. What I am suggesting, however, is counter to separatist sentiment—I claim that we should see what happens if we let the programmer talk about *both* procedure *and* representation.

PL/I is the latest language design to attempt suppression of representation. Yet, UNSPEC finally made its appearance. For a purist, the use of UNSPEC is as unacceptable as in-line machine language coding. Why, then, did UNSPEC slip in? The reason, I suspect, was two-fold: (1) to allow use of data types not already defined in the language, and (2) to allow a given storage cell content to be treated according to one data description at one point in a procedure and according to another somewhere else. It seems to me that the attempt to banish representation considerations from language design has, in this case at least, led inevitably to their sneaking in the back door in a quite unpalatable form.

The dilemmas posed by representation independence, UNSPEC, and the CELL construction in PL/I are unfortunate inheritances from past history, earlier forms of which occurred in FORTRAN. They resulted from our inherited belief that symbols are names of storage locations rather than names of entities assigned to them by the data organization. The crux of the situation is this: The programmer wishes to be representation independent to the extent that considerations of accuracy and economics allow. On the other hand, in some cases, he is definitely concerned with specific representations—for instance, in specification of procedures which convert data from one representation to another. To be so doctrinaire in language design as to insist on complete representation independence is to cut off one's nose to spite one's face. The language feature needed in the specific case at hand is the ability to say "I have a value for data item A in storage, and I now wish to use that as the value for data item B, which has a different data type. Furthermore, no matter how complex the structure and data organization for A and B, I *know* that the bit patterns for the two values are identical (I just told you so), so don't bug me about representation independence." In other words, the programmer must be allowed to pay his money and take his choice.

Language extension

PL/I has been lambasted both for having too many and for having too few data types available. The former point is made by many who have been unfavorably impressed by the sheer size of the code required in PL/I compilers, although this might with more justice be blamed on the amount of automatic data conversion required. The latter point is made by those who wish to handle data of more complex structure with less circumlocution. The suggestion has been made in many quarters that what we really need is fewer data types along with apparatus which will allow the user to define his own. I find myself in this camp, although I don't believe that the problem is by any means as trivial as is sometimes claimed.

Part of the problem, of course, lies in adopting a consistent view of what data are and finding suitable

*This is not my own point of view, needless to say. However, the notion of representation independence makes sense whatever one's philosophical bias.

methods for writing and storing data descriptions (including both representation and organization). The theory of data advanced above seems potentially adequate for these purposes; what is not so clear is how to link the data description apparatus to the code generation and optimization apparatus of the compiler. Galler and Perlis[5] have done some interesting work in this area.

Variable binding time

How much of the data description is stored explicitly, and where, is partially a matter of taste. It is largely determined by the language, language processor, and operating system one is working with at present. Classically, we have tended to use the data description at compile time and then throw it away. Moreover, much of the data description has been implicit in the compiler's structure; the programmer has had little explicit control. List processing is an intermediate case—processing variable data structures must be done interpretatively, even though the code for other processing can be compiled out. At the other extreme, interpretative operation tends to use the data description at execute time, over and over again. In the former case, the bet is that the program will not be recompiled too often and the data description will not have to change during execution, nor will the data structures. In the latter case, the bet is that less overhead will occur through interpretation than would be incurred by frequent massive compilations. Conversational processing tends in the latter direction.

But, in the case of all systems that have been designed thus far, the choice of binding time is pretty much cast into concrete by the system and language processor design, even though it is hard to believe that the choice can possibly be appropriate for all tasks. We should, I believe, seriously investigate the possibility of system designs that allow the individual user to make his own judgment of the proper tradeoff. This would necessitate explicit storage of data descriptions—a few systems have allowed this, notably the CL-I and CL-II systems.[6]

SUMMARY

In the first part of the paper, we have proposed a theoretical model for data and data processing. The model is a *system* of sets of *entities, values, data maps,* and *procedure maps*. The entities correspond to the objects in the real world about which data are recorded or computed. The data maps assign values to attributes of the entities; these maps are regarded as being sets of ordered pairs of entities and values, or *data items. Structural data* is a special type of data map where the value set is the set of entities itself; structural maps are composed of *pointers*. By introducing structural data maps and auxiliary entities, we can explain any data map as being composed of a structural data map followed by a map whose value is a quantity with as simple structure as we wish (for example, an integer). *Procedures* are operations on the data maps, producing new (or redefined) data maps. *List processing* procedures operate on structural data maps, or sets of pointers.

Data processing occurs in the machine realm, operating on objects which are mapped into the storage facilities of the computing system. We require, ideally, that such a system be a *representation* of the real or abstract system being modelled; this means that the representation map is one-one onto as regards the entities, values, data maps, and procedure maps and, further, that the representation map commutes with the procedure maps.

Access functions are maps whose values are entities; they are used by procedures to get access to the entities, and hence to the data. Access functions may involve use of structural data, but are principally a feature of the individual procedure. *Data organization*, on the other hand, is the way the structure of the data is mapped into the structure of the storage media.

Data description is a specification of machine data systems and representations; a *data type* is a fragment of data description, describing an entity and its applicable maps.

The first part of the paper, then, was a discussion of the nature of data and the relation between what it is and what we do with it. In the second part of the paper, these notions were used as a framework for discussing several issues of current interest.

The notion of machine independence, apart from its roots in practical needs, proceeds from the point of view that data processing takes place in the abstract realm and, hence, its results should be representation independent. This is an attainable goal only when our criterion for a representation can be satisfied economically. However, language design has tended to suppress the notion of representation to the extent that the programmer frequently cannot talk about it. I regard this as being a mistaken approach toward our practical goals; on the contrary, significant progress toward representation independence of results can only be made by making the notion of representation much more explicit in language designs than it is at present.

A second argument for emphasizing the role of data description in language design is related to the issue of language extension. A trend in language develop-

ment has been to expand the number of data types available to the programmer, at the expense of significant increases in compiler size and complexity, and without satisfying those who have a genuine need to use data types over and above those already available in a given language. The way out of this dilemma appears to lie in the design of languages with a limited number of data types (perhaps only one instance of each of the four generic types mentioned earlier) together with facilities enabling the user to introduce arbitrary data type definitions as needed.

Finally, it was argued that language and system design should make increased use of stored, explicit data descriptions. This will serve two purposes: First, it is a prerequisite for the design of systems which allow the user to strike his own economic balance between compilation and interpretation. Second, the extent to which current system designs achieve the goal of independence of procedure specification from data representation is due to the use of stored descriptive information, together with interpretation of the description, be it at compilation, execution, or some other time. Standardization of methods of data description may ultimately prove to be much more important than standardization of methods of data representation and procedure specification.

REFERENCES

1 W V QUINE
From a logical point of view
Harvard University Press Cambridge 1953

2 G L DAVENPORT E O DAVENPORT
The genealogies of the families of Cohasset, Massachusetts
Stanhope Press Boston 1909

3 A W HOLT
Proceedings of IFIP Congress 1965
Spartan Books Inc Washington D C 1966

4 D W MATULA
Base conversion mappings
AFIPS Conference Proceedings, Spring Joint Computer Conference vol 30 pp 311-318 1967

5 B A GALLER A J PERLIS
A proposal for definitions in ALGOL
Comm ACM 10 204 1967

6 T E CHEATHAM, JR.
Data description in the CL-II programming system
Digest of Technical Papers National Conference Association for Computing Machinery 1962

Dataless programming

by R. M. BALZER*
The RAND Corporation
Santa Monica, California

INTRODUCTION

A programmer using existing programming languages typically codes a problem by (1) defining it, then (2) analyzing the processing requirements, and (3) on the basis of these requirements, choosing a data representation, and finally, (4) coding the problem. Almost always, difficulties arise because necessary processing not envisioned in the analysis phase makes the chosen data representation inappropriate because of a lack of space, efficiency, ease of use or some combination of these. The decision is then made to either live with these difficulties or change the data representation. Unfortunately, changing the data representation usually involves making extensive changes to the code already written. Furthermore, there is no assurance that this dilemma will not recur with the new data representation.

The Dataless Programming System is designed to help alleviate this problem. All data and function references are expressed in a single canonical form, so that the alteration of a data representation has no syntactic effect on the program, but only affects the internal processing associated with the data or function references — i.e., changing the data representation does not affect the program's source statements. Unless the change in the data representation causes a change in data values (e.g., a change in precision), the values produced by the program will be unaltered. (Naturally, the behavior of the program in terms of efficiency, speed, and space required may be affected by any change in data representation.) Hence, the data representation can be specified after the program is written, and analysis of the problem definition and code production can be integrated into a single phase. The programmer should be able to construct his program in terms of the logical processing required without regard to either the representation of data or the method of accessing and updating. This concept we call "Dataless Programming."

Dataless Programming is more than a new language. It conceives of a program as the specification of a set of manipulations to be performed on a set of data values, and that this specification should be independent of the form in which these data values are represented. To achieve such independence, there must be a set of declarations that tell the programming system how to retrieve and store data values from the particular representation being used. The independence thus achived will allow the programmer (1) to disregard, while specifying the program, the details of data processing, memory space requirements, and matching of data representation to the processing done on it; and (2) to handle them, instead, during the data declaration phase.

The Dataless Programming Language is the embodiment of the above concept. It also helps simplify the construction and debugging of programs by providing high-level data-handling and control facilities (insertion, deletion, generators, bugs, search expressions, and implied qualifications of data references) and a STATE-statement (enabling a wide range of monitoring and tracing facilities to be specified).

Throughout the design of the language, an effort has been made to allow the programmer to express his intent more clearly through new program statements rather than in separate comments. That is, we have attempted to make the purpose of commonly occurring statement groups more apparent by combining them into an appropriately worded statement. Examples of this effort are the FOR-clause, search-expressions, and the extended IF-statement. The programmer can also increase the program's self-documentation through the

*This Research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Any views or conclusions contained in this paper should not be interpreted as representing the official opinion or policy of ARPA. This paper is a condensed version of the author's *Dataless Programming*, The RAND Corporation, RM-52 90-ARPA, July, 1967.

use of mnemonic names for data items which are then defined as functions (as has been done in the programming example in Sec. IX below).

II. The dataless programming language

The Dataless Programming Language is a high-level algebraic language which is an extension of PL/1 and uses PL/1 syntax notation* and syntax, except as noted. The language is based on a single canonical representation for all data and function references. This form is:

{collection-name|
function-name [(expression [, expression] . . .)]

If the name identifies a function name, the expressions, if any, are interpreted as parameters. If the name identifies a collection name, the semantic interpretation is that there exists a collection (or group) of data which has a common, unique name (the collection name). This collection has the property that, given a single integer index (computed from the expression, called the index expression, which follows the collection name), there exist algorithms which use the value of this index to operate on the data collection, performing the functions of accessing and updating the datum selected and inserting or deleting a datum from the collection. The system asumes that there exist such algorithms, determining what they are not from the form of the program's source statements but rather from explicit declarations supplied with the program. Hence, changing a collection from an array to a list does not change the source statements but merely changes a data declaration. Also, since function references are syntactically identical with data references, the user can change a function reference to a data reference, or vice versa, by merely changing a declaration.

Separating the data definition and its implied data-handling routines from the common appearance of data or function references in the source code provides the main power of the language. This feature allows the user to program in a top-down fashion in terms of the logical pieces of information necessary for the required processing. Once the user has completed this programming, he can determine for each collection what data representation is suitable for the processing involving that collection. He may also decide that a piece of information should not be supplied by a data reference on a collection, but should be supplied by a function through a calculation or series of calculations on other information—either data or further function references. The user may use the language's full power in defining either a function or the data-handling routine for his data representations, and so is able to program each part of his problem in a top-down fashion.

Logically, since the data-handling routines operate with a single index, the data representations are ordered lists. The system allows the following data representations (for which it provides the data-handling routines):

(1) ARRAY
(2) LIST (forward links only)
(3) DOUBLE LIST (list with both forward and backward links)
(4) RING (forward links only)
(5) DOUBLE RING (ring with both forward and backward links)
(6) STRUCTURE (as defined in PL/1)
(7) Structures of any of the above (e.g., arrays of lists of structures).

In addition, any other ordered area representation can be used by providing the necessary data-handling algorithms. These algorithms can consist of a call to an external procedure and can be written in any language desired (including assembly language). Hence, within the restriction of ordered data representations, the representations allowed are completely open ended. This class is large and significant, covering most currently used representations. Clearly, however, some representations (e.g., colored pictures) lie outside this class; and we have no ideas on ways to incorporate these into the Dataless Programming Language.

Since all data manipulations of a collection are performed by the data-handling routines, these routines can be altered to provide a powerful monitoring capability. By suitably altering the update routine for a data collection, the system can be notified any time a member of that collection gets updated. Hence, the system can perform any desired action as a response to the occurrence of a particular state of the program variable, i.e., it can handle STATE-statements. The form for these STATE-statements is:

{ON | WHENEVER} (boolean-expression) statement;

For implementation purposes, the boolean-expression is restricted so that all variables which effect its truth value must explicitly be part of the expression (enabling the system to determine which variables require checking), and the evaluation of the boolean-expression must not change the value of any of these variables (and care should be taken that it also does not change the value of any other program variables which affect the behavior of the program, so that its removal does not affect the program). Whenever one of these explicitly named variables is updated, the boolean-expression is evaluated and, if true, the associated statement is executed. As an example, if the STATE-statement

*As defined in Ref. 1, pp. 11-17.

ON (x = y | x = z) CALL print;

occurred, the data updating routines for x, y, and z would be altered so that this on-condition would be checked whenever either x, y, or z was updated. If the on-condition was met, the indicated action would be performed. Thus, a wide range of trace and/or debugging features can be specified. Also, a variety of special case monitoring of the program, which is non-debugging — but part of the desired processing — can be incorporated in the language.

The language provides the following data-handling statements (small single letters will be used to denote data collection names):

(1) DELETE region;
(2) INSERT region {BEFORE | AFTER} x (index-expression) [AND MAKE CURRENT];
(3) REPLACE region BY region;
(4) ADD region TO x [AND MAKE CURRENT];

where a region is defined as

collection-name (index-expression)
[TO collection-name (index-expression)]

and specifies a continguous inclusive set of collection members (the collection names must be the same and the indices must ascend in order). (The AND MAKE CURRENT option is explained below in Sec. III.) The REPLACE and ADD statements are defined in terms of the INSERT and DELETE statements. REPLACE is equivalent to DELETE followed by INSERT BEFORE, and ADD is equivalent to INSERT AFTER the last member of the collection. The above statements are defined for all data representations allowed in the language; e.g., insertion into an array is defined as increasing by 1 the index of all members with indices greater than or equal to the value (i) of the inserted index, and the storing of the value of the inserted datum as the new *ith* member of the array. (The problems concerning the maintenance of data collections and of pointers to members of these collections are discussed in Sec. VI below.)

Two facilities are provided for sequencing through a data collection. The first is the FOR-clause which sequences through a data collection searching for all members which satisfy a specified condition and, as each one is found, causes a single statement or a group of statements to be executed. Thus, one cycle of an "iteration" is performed for each member of the subset of the data collection which satisfies the specified condition. This facility essentially combines a DO-statement (to sequence through the data collection) with an IF-statement inside the range of the DO-statement to test the given condition.

The second sequencing facility: (1) sets up a search through a data collection, in the same way as a FOR-clause; (2) searches for the first member of the data collection which satisfies the given condition; and (3) causes a placemaker to point to that member. This placemarker can be used for processing the selected member. When the next member of the sequence is desired, it is explicitly requested, causing (1) the data collection to be searched for the next member which satisfies the previously specified condition, and (2) the setting of the placemarker to point to the newly selected member. This facility generates, upon request, the next member of a collection which satisfies a given condition. Hence, the facility is called a "generator," the statement which sets up the sequencing a "GENERATE-statement," and the placemarkers "generator-variables." The generator method of sequencing differs considerably from the FOR-clause method in that (1) it is not associated with a particular statement or group of statements, and (2) is not automatic but occurs only on request. The same code is not necessarily used to process all members in the sequence, and the same code can be used for different data-collection sequences. These generators can operate independently of each other, and more than one can be sequencing through a data collection simultaneously (if two or more generators are sequencing through a data collection, and the processing done in connection with one of them alters the data collection, the set of selected members of the other generator(s) may also be altered, as the current state of the data set is used in the determination of the next member to be selected).

The FOR-clause is used to control the execution of a single statement or group of statements for selected members of a collection. Its syntax is:

FOR {ALL | EACH | EVERY} collection-name
[iterative-specification]

where an iterative-specification is defined as

[IN THE DOMAIN specification [, specification] . . .]
[SUCH THAT (Boolean-expression)]

and a specification has the form

expression-1 [TO expression-2] [BY expression-3]
[WHILE (expression -4)]

The precise semantics of a specification are given in the PL/1 specifications,[1] but basically correspond to the following: a region is specified (by expressions 1 and 2) through which an index is incremented in steps of size expression 3; and the WHILE-clause specifies a condition which, if not met, causes the termination of that specification. Notice however that contrary to the DO-clause, the index is not specified; it is not needed and is supplied automatically by the system.

The FOR-clause is associated with a single statement by inserting the FOR-clause before the statement's ter-

minating semicolon (e.g., x = y FOR ALL x SUCH THAT (x < 5);), or with a group of statements by preceding the statement by:

FOR-clause DO;

and following them by

END;

The FOR-clause causes execution of the associated statement or group of statements for all members of the collection in the specified domain for which the boolean-expression is true. The iterative specification has the same interpretation as in PL/1. If no iterative specification is given, then all the members of the data collection, taken in ascending order, are used as the domain.

Generators are used to give the programmer explicit control of the sequencing through a data set. A new variable type—generator-variable—is introduced for this facility and is used to obtain the successive members of the collection. These variables allow more than one generator to be operating on a data collection at once, and also allow one generator variable to sequence through two or more different data collections during program execution (but only one at a time). The syntax for generator statements is:

GENERATE THROUGH collection-name USING generator-variable [AND MAKE CURRENT] [iterative-specification];

This statement sets up the sequencing implied by the iterative-specification as defined in the FOR-clause (or a sequence of the successive members of the collection taken in ascending order if the iterative-specification is not specified) and establishes the generator-variable as a synonym for the first member of the collection selected by the sequence. (The AND MAKE CURRENT option is explained below in Sec. III.)

The programmer explicitly instigates iteration by execution of the GET NEXT statement:

[GET] NEXT (generator-variable [{AND | BUT DO NOT} MAKE CURRENT]; [OTHERWISE statement]

This causes the generator-variable to become a synonym for the next member of the collection selected by the GENERATOR-statement. If no other members exist for the generator and an OTHERWISE-statement (which can be a block or group) is present, it is executed as an ON-unit.* If not, a new condition—the END GENERATOR condition—is raised and can be handled by an appropriate ON-unit. (The maintenance of the correspondence between a generator-variable and a collection member in a dynamic environment is explained in Sec. IV below.)

Search-expressions cause a data collection to be searched for a member which satisfies a condition. Either this member or its index is the value of the expression. The syntax of a search-expression, which can be used anywhere an expression can, is

[INDEX OF] numeric-specification collection-name [IN THE DOMAIN specification] SUCH THAT (boolean-expression)

where specification is defined as in the FOR-clause, and numeric-specification has the format:

{FIRST | SECOND | LAST | {decimal-integer (expression)} {ST | ND | RD | TH} }

Examples:

10TH
21ST
(x+3)RD

The numeric-specification specifies a value i, and the value of the search expression is the *ith* member (or the index of the *ith* member if INDEX OF is specified of the data collection which satisfies the Boolean-expression with the order of iteration specified by the iterative-specification.

The statement following a statement containing a search-expression can begin with the keyword OTHERWISE. This statement is executed as an ON-unit** if and only if a value cannot be found for the search expression.

If the OTHERWISE option is not specified and the search-expression does not produce a value, a new condition, SEARCH FAILURE, is raised and can be handled by an appropriate ON-unit.

Many times a search-expression is used only to find out whether or not a member of a collection exists which satisfies a certain condition. To make the intent of this use of the search-expression more apparent, the syntax of the IF-statement has been extended as follows:

IF {boolean-expression | THRERE [{DOES ' DO} NOT] {EXIST | EXISTS} {A | AN | number | (expression)} collection-name SUCH THAT (Boolean-expression)} THEN statement [ELSE statement]

The semantics of the IF-statement remain unchanged;

*ON-units and conditions are fully explained in Ref. 1, pp. 79-84, but basically consist of the following: When such conditions as fixed point overflow, end of file, or END_GENERATOR occur, the program execution is interrupted. If an ON-unit has been specified to handle a condition which has occurred, then it is executed. This group of code can take corrective action and return to the program, print out an error statement, or terminate the program. If no ON-unit has been specified, the software system takes some default action (usually terminating the program). ON-units can be specified for any occurrence of a condition or only for those occurrences related to a specified set of variables. Thus the action taken can be dependent on which variable was related to the occurrence of the condition.

**For a discussion of ON-units and conditions, see Ref. 1, pp. 79-84.

i.e., the statement following the THEN is executed if and only if the condition specified between the IF and the THEN is true; and the statement following the ELSE, if present, is executed if and only if the condition is false.

Examples of the extended IF-statement are:

IF THERE EXISTS A y SUCH THAT (y < 10)
THEN GO TO z;

IF THERE DO NOT EXIST 10 y SUCH THAT
(y > 0)
THEN RETURN (0); ELSE RETURN(1);

III. Hierarchical data references

For data representations which are hierarchical, references to the elements can be fully specified by using the following form, where n is the level of the element being referenced:

Collection-name $_n$ (index $_n$) OF collection-name $_{n-1}$ (index $_{n-1}$) OF . . . OF collection-name $_1$ (index$_1$)

For example, consider a list, x, of arrays, y, where the *ith* member of the *jth* array is referenced by

y (i) OF x (j)

This notation assures unambiguous data references but is: notationally burdensome; restricts our reuse of the variables used in higher level indices; and necessitates referencing through each level of the hierarchy. Instead, it would be convenient to reference data relative to some member of a collection. Once a user has decided to talk about the *jth* member of the collection x, he would like to be able to reference the *ith* member of collection y by writing

y (i)

This facility is achieved by introducing a new statement which allows the programmer to specify that a particular member of a collection is to be made a reference point for further data specifications. This is called making the member current. One (and only one) member of each collection can be made current. The form of this statement is:

MAKE collection-name (index-expression) CURRENT;

The specified member of the collection is made current.

Each FOR-clause saves the current member of the data collection being iterated through, and each iteration of the loop makes the newly selected member current. Normal termination of the loop (the iteration has been completed) causes the saved member to be made current again. If the loop is terminated by a transfer (GO TO), the current member remains current and the identity of the previous current member is lost.

Whenever a reference is incompletely specified, the system will supply the necessary current member (or members) of the missing collection (or collections) to complete the specification of the reference. This is done in the following way:

(1) If the highest level collection in the reference is not completely specified, then the current member of the collection at the next higher level is used to complete this part of the specification. Thus, in the previous examples, a reference of "y(i)" will cause the system to complete the specification by supplying the current member of collection x.

(2) If there are any missing collections which are intermediate in level between the highest and lowest collection specified in the collection, the index of the current member of those collections is used as the index of those collections to complete the specification. For example, given a five-dimensional array (x1, x2, x3, x4, x5), the reference "x4(2) OF x1(10)" would be completed by the system and become "x4(2) OF x3(CURRENT) OF x2(CURRENT) OF x1(10)." (CURRENT is a function whose value is the index of the current member of the collection in whose index expression the function reference appears.)

Notice that the above completed specification is different from the reference "x4(2)" (which would be completed by rule 2) above to "x4(2) of x3"), and the current member of x3 does not necessarily have to be a subpart of x1(10).

Any data reference may require none, one, or both of the above rules to be completed. For example, "x4(2) OF x2(5)" requires both rules to be completed as "x4(2) OF x3 (CURRENT) of x2(5) of x1." By use of these mechanisms, data references can be specified relative to a base determined by the current member of a data collection. Generator-variables can also be used for relative data specifications by specifying the generator-variable as the relative base. For example, if "gen" were a generator-variable sequencing through data collection x, then "y(i) OF gen" would refer to the *ith* member of whichever member of x "gen" was set to. This method of hierarchical references is similar to the use of "bugs" in L6[2] and to the concept of "current" in APL[3].

Since generator-variables can be set to an individual member of a collection (e.g., "GENERATE THROUGH x USING gen IN THE DOMAIN 5"; causes "gen" to be set to the fifth member of the collection x), the full power of the bug concept of L6 is available. To make this cause of generator-variables easier, the following statement has been introduced:

SET generator-variable TO collection-name (index-exp);

This causes the specified generator-variable to be set to the specified member of the named collection (any attempt to execute a GET NEXT statement for

this generator variable will raise the END-GENERATOR condition).

To extend the use of the current concept to generators, options have been included which allow the programmer to specify whether the members generated should or should not be made current. In the GET NEXT-statement, he can specify whether the members generated through the use of that particular GET NEXT-statement should be made current or not. If neither option is specified, then the GENERATE-statement is used for the determination. If the AND MAKE CURRENT option is specified, the generator members will be made current; otherwise, they will not.

Similarly, the programmer can specify in an INSERT or ADD-statement that the inserted or added member is to be made current. (If a region is inserted or added and the AND MAKE CURRENT option is selected, the last member of the inserted or added region is made current.)

To extend the programming facilities and/or for notational convenience, the following built-in functions have been defined—where a collection-member-expression can be a collection member reference, a search-expression, a generator-variable, or a function reference which returns a collection member as its value.

NEXT (collection-member-expression),

PREVIOUS (collection-member-expression), and

INDEX (collection-member-expression)—the value of the function is the next or previous member, respectively, of the collection specified in the expression from the specified member or the index of the specified member of the collection. An OTHERWISE-statement can follow a statement including these functions. This statement is executed if and only if the desired member does not exist. If an OTHERWISE-statement is not employed and the desired member does not exist, the NO_NEXT, NO_PREVIOUS, or NO_INDEX condition is, respectively, raised.

NUMBER (collection-name) returns the number of members in the specified data collection.

LAST and CURRENT return, respectively, the number of members in the specified collection and the index of the current member of the specified collection. These functions can only be used inside an index-expression, and the collection referenced is the one to which the index-expression applies.

NEW [(collection-name)] a new member of the specified collection is created; and, if initial values have been specified, it is initialized. This function can only be used in an ADD or INSERT-instructions; and if an operand is not specified, a new member of the collection being added to or inserted into is created. This function together with the ADD and INSERT-statement constitutes the collection-building capability of the language.

IV. Data definition

The individual members of a collection that are not themselves collections can be any of the forms defined in PL/1 for data elements. Arrays, lists, and rings are defined by inserting the appropriate keyword (ARRAY, LIST, DOUBLE LIST, RING, or DOUBLE RING) after the collection-name in the PL/1 definition. Structures are defined as in PL/1. Thus an array of lists of structures would be defined as:

```
DECLARE
   1  department (20 ARRAY,
      2  people LIST,
         3  name CHARACTER (20) VARYING,
         3  man-number BINARY FIXED,
         3  projects LIST CHARACTER (10);
```

For those data representations that cannot be defined using the system-defined representations, the definition is supplied by providing programs which handle the necessary manipulations on the data representation. The form of this definition is:

```
Collection-name ACCESSED BY program 1;
                UPDATE USING program 2;
                INSERT USING program 3;
                DELETE USING program 4;
```

The programs used in the data definition can contain any of the features provided by the system. In addition, they can include, where applicable, references to the addresses of an operand and the contents of an address. Two system functions are provided for this purpose:

ADDRESS (variable)

and

VALUE (location, from)

where location specifies where the desired value is, and form specifies the transformation to be used to extract a value from this location. Typical forms are (with the IBM 360's in mind):

```
BF  = sign + 31 numeric bits;
HBF = sign + 15 numeric bits;
QBF = sign + 7  numeric bits;
QL  =        8  numeric bits;
QB  =        8  logical bits.
```

These same forms could be specified in the left-hand side of an assignment statement to indicate the form in which a value should be stored:

```
y = VALUE (ADDRESS (x) + 5,HBF) +3;
```

the 16 bits at location (address (x) + 5) are treated as a sign + 15 numeric bits and are added to the number 3 and the result is stored in variable y

```
x-2 IN FORM BF = ADDRESS (y);
```

the location (address (y)) is stored in location (x-2)in form BF

When the system passes collections as parameters to a subroutine, it also passes the necessary data-handling routines (either the standard system routine or the user's routine) so that the subroutine can operate on any allowable data collection. Such routines are data-representation independent, and libraries of them should provide a flexible programming environment. One major problem not satisfactorily solved is finding a method that allows the user to change the form of the variables of a collection (such as from character strings to floating point numbers) and still use the same subroutine to process both forms. Present plans for handling this have the user:

(1) declare those parameters which can be of different forms to the 'FREE,'
(2) perform form checking on all operations involving these FREE parameters,
(3) utilize the correct routine to handle them properly.

The user would also have to specify in the calling program which parameters to the subroutine were FREE so that their forms could also be passed.

V. Further language features

In defining the Dataless Programming Language, several other features seemed desirable that were not currently available in algebraic languages in general—and specifically not in PL/1. These features are included here* (although they do not directly relate to the main goal of separating data description from program description):

(1) a source-level execute command which causes the named statement or group of statements to be executed, and allows the passing of parameters for this execution:
(2) a compare statement which stores the result of a comparison in a cell associated with the label of the compare statement, and which allows this cell to be subsequently interrogated and, if desired, modified;
(3) the CASE function of LISP 1.5, which permits an expression to determine which of a series of expressions should be used as the value of the function.

VI. Maintaining pointers in a dynamic environment

There are two types of pointers in the Dataless Programming Languages: the generator-variables, and the current member of each data collection. Both types point at members of data collections, and these pointers must be maintained while the data collections are altered.

There are two types of alterations which can affect these pointers. The first is an insertion or deletion from the data collection, causing a change in the location of the members of that collection (an insertion or deletion from an array has this effect). The second is the deletion of a member being pointed to. These pointers can be maintained by keeping a list for each data collection of all pointers which point to members in that data collection. While moving or deleting a member (and its submembers), the system can check for pointers which refer to the affected members (and submembers), and take appropriate action. This action for movement is merely the repositioning of the appropriate pointers so that they point to the member's new location. In the case of deletion, the action taken is more complex. First, the affected pointers are set to a state called "undefined" (all pointers—generator-variables and current member of all data collections—are initially set to this state), which will cause any subsequent attempt to reference the deleted member to raise the UNDEFINED_POINTER condition that can be handled by an appropriate ON-unit. Secondly, internal pointers to the next and previous members of the collection (if they exist) are associated with the undefined pointers so that they can subsequently be used to move to these members. (If these members do not exist, an attempt to use these internal pointers will cause the NO_NEXT or NO_PREVIOUS condition to be raised.) These internal pointers are also maintained through any subsequent alterations of the data collection.

VII. Implementation

Present plans call for an implementation of the Dataless Programming Language, in a restricted form, through an editing program utilizing Leavenworth's Syntax and Function Macros[4] (which puts the program into standard PL/1) plus a set of run-time routines. This method will impose certain restrictions (e.g., lists and rings must be implemented as arrays because the list processing capability of PL/1 is not available) and will be detrimental to program efficiency. But it will enable running programs written in Dataless Programming at a relatively small cost. With experience, the language can be improved, and plans made for a full implementation. Although most of the capabilities of Dataless Programming can be achieved by placing an editor between the program and the PL/1 compiler, this is no way alters the fact that Dataless Programming provides a radically different view of the programming environment than does PL/1.

*These and other features will be fully described in a paper now in preparation.

VIII. Expected advantages and difficulties

Programming should be simplified because it can be constructed top-down in terms of the logical processing required. The problem of data representation can be left until this programming has been completed; thus, a more rational decision can be made concerning an optimal representation. Because of this separation, the programmer should be able to think through his problem better.

The data-handling features incorporated to handle all data homogeneously put the language into a more canonical form, make it more mnemonic, improve its readability, and increase its self-documentation. This improved readability will undoubtedly help reduce the number of programming errors in the original programs, and make finding and correcting the remaining errors easier. All this, in some sense, increases the "high-levelness" of the language.

Since Dataless Programming takes care of passing the necessary data-handling routiness to subroutines or functions libraries of routines which are data-representation independent (within the basic ordered-list restriction of the language) can be created for use in a general programming environment.

Also, since data representations can be easily changed, it is possible to determine experimentally which is best for the given program, i.e., with reference to the Dataless Programming system. Even so, this would provide one objective measure for different data representations from which criteria might be generalized and developed for the choice of representations.

The main difficulty with the system is the level of efficiency. Inefficiencies come from two sources: (1) the separation of the data-handling routines from the program and from the homogeneous manner in which all data are handled; (2) since data treated homogeneously, the programmer cannot take advantage of the special properties of one representation in writing a program without having already decided what representation will be chosen. It is hoped that once the program is debugged there will be ways to remedy some of the inefficiencies.

There are several other difficulties and shortcomings of Dataless Programming, including the incomplete separation of data description from program description, and the lack of a method of extending the allowable set of data representations beyond ordered lists. Also, the language is now a hodgepodge of all the facilities considered desirable for either their processing capabilities or convenience. As such, it is not a smooth, polished, well-integrated system.

The essential considerations, however, are how easy and natural is it to learn and program in the Dataless Programming Language, and how much more powerful it is than existing languages.

IX. Dataless programming example

In the following example, the problem statement is followed first by the uncommented program and then by the set of comments pertaining to the program. The numbers at the start of each line of the program are not part of the program, but are used to associate comments with statements. Notice that almost all comments explain the functioning of the language statements; little commentary is needed to explain the processing, i.e., once the functioning of the various language statements has been mastered, the program itself becomes mnemonic because the intent of the programmer is usually apparent from the language statements themselves.

The example chosen is a problem which appeared in he article, "APL—A Language for Associative Data Handling in PL/."[3] Since APL and Dataless Programming share many similar features, the problem affords a means of comparison. This example was also chosen because it illustrates many of the facilities of Dataless Programming, including the use of functions as if they were data, the automatic passing of the current member of a collection, the use of a generator-variable as a bug, and the use of search expressions (with the OTHERWISE option), iteration statements, and the collection-building facility.

The problem is to compute the starting and ending dates for a set of jobs. The input data consists of a set of entries, one for each job. Each entry consists of the job's jobname and its duration, followed by a list of all jobs that must be completed before this particular one can be started. It is assumed that these dates are in a form suitable for PL/1 list-directed input.

```
1.          ON ENDFILE GO TO compute;
2. read:    GET LIST (ident);
3. locate:  MAKE 1ST job SUCH THAT ident = jobname CURRENT;
4.          OTHERWISE DO;
5.              ADD NEW TO job AND MAKE CURRENT;
6.              jobname = ident;
7.          END;
8.          GET LIST (duration);
9.          SET present_job TO job;
```

```
10. read__predecessor:  GET LIST (ident);
11.       IF ident = '' THEN GO TO read;
12.       EXECUTE locate;
13.       ADD INDEX (job) TO predecessor OF present__job;
14.       GO TO read_predecessor;
15. compute:
16.       FOR EACH job SUCH THAT (all_predecessors_finish_known
                ϵ finish date is unknown) DO;
17.            start_date = max_predecessor_finish + 1;
18.            finish date = start date + duration;
19.            GO TO compute;
20.            END;
21.       IF THERE EXISTS A job SUCH THAT finish_date_is
               unknown THEN
22.            PRINT LIST ('incorrect data')
23.  ELSE
24.            PRINT LIST ('satisfactory run');
25.  DECLARE
26.       all_predecessors_finish_known BIT (1)
27.         ACCESSED BY
28.               IF THERE EXISTS A predecessor SUCH THAT
                  (finish_date_is_unknown (predecessor)
                  THEN RETURN ('O'B);
29.         ELSE RETURN ('1'B); ,
30.       finish_date_is_unknown BIT (1)
31.       ACCESSED BY
32.         IF_finish__date = 0 THEN RETURN ('1'B);
33.         ELSE RETURN '0'B);,    0
34.         max_predecessor_finish BINARY FIXED
35.  ACESSED BY
36          DO;
37.         x = 0;
38.         x = MAX (x, finish_date (predecessor) ) FOR
                        EACH PREDECESSOR;
39.  END;
40.       1 job LIST,
41.            2 jobname CHARACTER (20),
42.            2 start date BINARY FIXED INITIAL (0),
43.            2 finish date BINARY FIXED INITIAL (0),
44.            2 duration BINARY FIXED,
45.            2 predecessor LIST BINARY FIXED;
46. DECLARE
47.            ident CHARACTER (20),
48.            present job GENERATOR,
49.            x BINARY FIXED;
```

Comments

1 Sets up an ON-Unit that will cause transfer of control to the statement labeled "compute" when the end of the input file has been reached.

2 A PL/1 input statement that causes the next value in the input file to be assigned to the named variable ident.

3 Find and make current the job that has a jobname equal to the value of ident.

4 An OTHERWISE-clause. These statements are executed if and only if the search in statement 3 failed, i.e., if there did not already exist a job that had a jobname equal to the value of ident.

5 Creates and initializes a new job, adds it to the list of jobs and makes it current.

6 The jobname of the newly-created job is set equal to the value of ident.

8 The next value in the input file is assigned to

the named variable duration (the reference "duration" is incomplete and is completed by using the current member of data collection job, hence the duration of the current job is the variable referred to).

9 The generator-variable present job is set to refer to the current job.

10 The next value in the input file is assigned to the variable ident. This value is the name of a predecessor of the current job.

11 Test for the end of the list of predecessors. If the value of ident is NULL, the end of the list has been reached and control is transferred to the statement labeled read to process the entry of the next job.

12 This is an additional feature that causes the specified statement (the statement labeled locate together with its associated OTHERWISE-clause, i.e., statements 3-7) to be executed, after which execution continues with the next statement (statement 13). This group of executed statements (3-7) makes current the job whose name is the value of ident, and if this job does not already exist, creates and initializes it.

13 The index of the job which is a predecessor of the job referred to by present job is added to the collection of such predecessors.

14 Get another predecessor.

15 The input phase has been completed; calculate the start and finish dates for the jobs.

16 Sequence through the collection of jobs, selecting those which satisfy the given condition. (The condition is specified in terms of two items defined in the data declaration.) Notice how the details of finding the next job to be processed are removed from the program and can be stated in terms of the logical requirements of the process, i.e., to process a job the finish data of all its predecessors must be known and its own finish date must be unknown.

17 Calculate the start date of the current job (a job selected by statement 16 above) by use of a data item defined in the data declaration. Again, notice how the processing has been expressed in terms of the logical processing required.

18 The finish date of the current job is set equal to the start date of this job plus its duration.

19 Begin the search again.

21-24 Test for the satisfactory completion of the program and print conclusion. If any jobs have an unknown finish date, then the data used in the program must be incorrect.

25-29 Definition of all predecessors finish known as a function.

28 The collection of predecessors of the current job is searched for one for which the finish date of the job whose index is the value of predecessor is unknown. If such a predecessor exists the finish date of all the predecessors of the current job is not known and the bit string '0'B is returned.

30-33 Definition of finish date is unknown as a function.

34-39 Definistion of max predecessor finish as a function.

40-45 Definition of the structured collection named job.

46-49 Definition of the remaining variables.

ACKNOWLEDGMENTS

I am indebted to J. C. Shaw and Dr. Allen Newell for their comments and suggestions on the topics contained in this paper.

REFERENCES

1 IBM operating systems/360 PL/1 language specifications
IBM Form C28-6571-3 IBM Corporation White Plains New York 1966

2 K C KNOWLTON
A programmer's description of L[6]
Communications of the ACM
vol 9 August 1966

3 G G DODD
APL—A language for associative data handling in PL/1
AFIPS vol 29 Proc FJCC Spartan Books Washington DC 1966

4 B M LEAVENWORTH
Syntax macros and extended translation
Communications of the ACM vol 9 no 11 November 1966

5 A NEWELL
Information processing language-V manual
Prentice-Hall Englewood Cliffs New Jersey 1961

6 P ABRAHMS et al
The LISP 2 programming language and system
AFIPS vol 29 Proc FJCC Spartan Books Washington DC 1966

7 J C SHAW
JOSS: A designer's view of an experimental on-line computing system
The RAND Corporation P-2922 August 1964

8 *Revised report on the algorithmic language ALGOL-60*
Communications of the ACM vol 6 no 1 pp 1-17 1963

PLANIT–A flexible language designed for computer-human interaction*

by SAMUEL L. FEINGOLD
System Development Corporation
Santa Monica, California

INTRODUCTION

Time-sharing has brought about a new age in computer usage: With the advent of time-shared systems, on-line use of the computer will become economically feasible for the average user; and on-line with him will be a wealth of software utility support never before available. No longer faced with a 24-hour turnaround—the average turnaround time in some existing time-sharing systems is a few seconds or less—he can perform program research, development, and checkout with ease undreamed of heretofore.

But time-sharing also has a price: It demands that we do something about an area that has previously been considered too expensive to address properly. We might call that area Computer-Human Interaction (CHI). Before time-sharing, only fragmented CHI could take place unless large blocks of computer time (costing large sums of money) were dedicated to individual users. And during such costly interaction, most of the time the computer was "idling," waiting for the user to make an input. Now that time-sharing provides the capability for many users to converse with the computer simultaneously and for extended periods of time, we are faced with the opportunity—and the need—to refine this interaction.

Many programs have now been written for time-shared use—programs for computer-assisted instruction, computer-based vocational counseling, computer-assisted training, interviewing for medical history, etc. These might all be described as interactive programs: The user may spend an hour or more interacting with the program at the teletype or display console.

The following discussion describes a language (and program) called PLANIT that has proven extremely useful for the preparation and execution of interactive programs.

*The work reported herein was sponsored by SDC and the National Science Foundation under Grant GY-371.

Background of the project

PLANIT (**P**rogramming **LAN**guage for **I**nteraction and **T**eaching) is a system employing a flexible language designed for CHI. PLANIT provides for the designer a powerful and flexible tool for entering communication material into the computer, for modifying the material, for presenting it to the user, and for prescribing the behavior of the computer as a function of the user's current response and past performance. In addition, the language is simple enough to allow a nonprogrammer to use it easily.

The program and language were designed at System Development Corporation by a study team from the Research and Technology Division's Education and Training Staff. The purpose of their project (supported partially by an NSF Grant and partially by SDC's independent research program) was, in general, to explore the use of the computer in the teaching of statistics and, in particular, to develop a set of computer-based instructional materials. Initial efforts produced a first version of the laboratory portion of a course in statistics. A program, called STAT, was coded in a high-level language, JOVIAL.[1] This first program, tested on nine graduate students in psychology at the University of California, Los Angeles, was useful in pointing out a number of problem areas in the material. Moreover, one significant obstacle to our work efforts soon became apparent during the test runs: The capability for changing parts of the lesson during test runs was poor.

There were three team members involved in the development of STAT—a statistician, an educator, and a programmer. Typically, when any one of us wanted a change made, he would first have to find the other two, then explain to them and convince them. Then the material would have to be recoded into JOVIAL, recompiled into another program, and finally checked out. (Meanwhile, often the students were there wait-

ing, being stalled in the conference room.) At any rate, this first go-around showed us that what we needed was a general-purpose program that would assist a nonprogrammer in preparing instructional material for statistics. In addition, the user must be able to enter the material easily and be able to change it as easily. We feel we have succeeded in our attempt to design such a program. (For example, STAT took three men, 14 man months, to produce. We believe that one man—a knowledgeable nonprogrammer—could empty STAT into PLANIT in from two to three months, after he is familiar with the lesson-building and calculation (CALC) modes of PLANIT.)

Related projects

Prior to building PLANIT, the author toured the country in search of a program or ideas that could satisfy the demands of the STAT project. Some of the more interesting programming languages or systems seen some 20 months ago are described below:

A. PLATO (**P**rogrammed **L**ogic for **A**utomatic **T**eaching **O**peration), a program developed by a study team at University of Illinois.[2] PLATO III uses a CDC 1604 computer (48-bit words, 32K words core). Most impressive was PLATO's speed of operation and its on-line editing capability. PLATO's slide-oriented system allows for very fast interactive speed. The access time to any of a possible total of 240 slides is 1/10 second. The slides are ordinary transparencies mounted on transparent plastic sheets, 120 slides to a sheet. The system can take a maximum of two such sheets in the central slide selector. Through CATO (a language developed for writing courses to run under the PLATO system) it is possible to write sophisticated instructional logic.

The system, however, has some features that made it unacceptable for our purpose. The limit of 240 slides imposes a serious restriction on the number and combination of courses going on at the same time without previous notice. Furthermore, though on-line editing is easy for minor changes, any substantial modification of the lesson requires a recompilation of the course. Also, while CATO allows complete freedom for structuring complex instructional logic, the language (FORTRAN-like) probably requires a good deal of training and experience for a user to become proficient in it.

B. COURSEWRITER, an IBM program in use at Pennsylvania State University, Florida State University, University of California at Irvine, UCLA, and University of Michigan.[3] A number of COURSEWRITER systems have been used throughout the country. The systems vary in internal design from interpreter to compiler and employ various computers including the 1410-7740, 1401, 1440, 1460, and 1500 operating systems.

COURSEWRITER, which is being marketed by IBM, is by far the largest operating system in existence. It was designed to be used in a "large" sense, i.e., for many courses simultaneously by many students. The system provides a set of routines for accommodating and evaluating typed responses. This feature is a step toward the analysis of constructed responses, and is ideally suited to lengthy sequences for shaping spelling behavior. (The incorrectly spelled input word is echoed by the system with blanks in place of incorrect characters.) The basic language, COURSEWRITER, is fairly easy to use for lesson preparation, but it has no real calculation capability; and any deviation from the basic language set for special functions not already included requires a knowledge of AUTOCODER.

C. SOCRATES, developed at the Training Research Laboratory, University of Illinois.[4] The program ran under an IBM 1620 computer and IBM 1710 control system dedicated to evaluating multiple-choice responses. This was a small system with no general-purpose software language for author input. This program, to the best of the author's knowledge, is no longer being used.

D. MENTOR, a computer language for programmed discourse developed by Bolt Beranek and Newman Inc., of Cambridge, Massachusetts. It is, in this author's opinion, one of the best languages designed for interactive discourse. For our purposes, however, we needed a sophisticated calculation capability that could operate on student records of past performance and could also be used by students in working exercises.

E. BASIC,[5] developed at Dartmouth, and TELCOMP,[6] developed at Bolt Beranek and Newman Inc. These programs are similar in nature: Both are unsuitable for use by nonprogrammers, since they employ high-level language. They are used to program completed routines allowing the student to explore the computing potentials of these languages.

Failing to find a program that satisfied our particular needs, we designed our own. This eventually became PLANIT.

Description of PLANIT

The initial design started in January of 1966; by June PLANIT was operational. It was written in JOVIAL and used the IBM AN/FSQ-32V computer via an interactive console under the SDC time-sharing system.

Our original purpose was to have a language that would enable a user to build instructional material in

statistics. However, since computer-based instruction in statistics requires so much flexibility and power, we discovered when we were through that we had more than we expected: We had a language designed to assist an author in building not only mathematical and related material, but such courses as history, spelling, psychology, etc. We have since added a few additional features, so that we now have a multipurpose language designed for CHI, of which lesson building is a subset.

PLANIT comprises not only a language but also a program, developed for time-shared use. The system operates in four modes: Lesson building, editing, execution, and calculation. The first two modes permit the teacher to construct and edit lesson frames in various formats and store them in designated sequences for later presentation to the student in the execution mode. The calculation mode is particularly oriented to mathematical subject matter and can be used as a calculation aid for the teacher (when building the lesson) or the student (when performing the lesson). While the student has access only to modes EX (execution) and CALC (calculations), the teacher or lesson designer (LD) may use all four modes.

- *Calculation Capability.* PLANIT has an on-line calculation capability that allows either the LD or the student to perform calculations involving trigonometric functions, elementary algebraic functions, and matrix declarations. In addition, the calculation functions of PLANIT can be tied in with the lesson: The LD can request the student to compute some data and can specify that the student's answer be compared with the results of evaluating a previously defined function. The LD can use the CALC mode to define the function when preparing the lesson; during lesson execution the CALC mode can be used again to "test" the student's answer.
- *Criterion Branching.* The PLANIT language allows the LD to specify conditions for branching based on the student's performance over any portion of the lesson.

  Conditions for branching may include:
  1. Response latency on any one answer or group of answers.
  2. Number of errors made on any group of questions.
  3. Help received from the CALC mode (functions used or not used).
  4. The actual path taken through the lesson up to this point.
  5. Any combination of the above four points.
- *Service Routines.* PLANIT also provides the following service functions for evaluating student answers that depart from the anticipated responses specified by the LD:
  1. PHONETIC Comparison. This routine gives the student credit for his answer even though it is spelled incorrectly as long as it is phonetically equivalent to one of the LD's answers. For example, PLANET and FONETIC are acceptable wrong spellings for PLANIT and PHONETIC.
  2. KEYWORD Match. This routine instructs PLANIT to search for a set of words in the student's response as the KEYWORDS of his answer. Furthermore, if the lesson designer wishes, he may specify that the answer will be evaluated as correct only if these keywords appear in a prescribed order.
  3. FORMULA Equivalent. This routine will allow the student credit for his answer as long as it is one of a subset of expressions algebraically equivalent to any one of the LD's answers. For example, if one of the LD's anticipated answers is 5/9* (F-32), then (5*F-160)/9 or 5*(F/9)-160/9 or (−32*(5)+5-*F)/9 would be equivalent and therefore acceptable. The LD can have any combination of these three routines turned off and on during lesson execution (even between actual occurrences of anticipated answers during student interaction with the lesson).

Illustrations

A lesson is composed of a set of frames; frames are composed of groups; groups are composed of lines of information such as textual material, questions, anticipated answers, actions, etc. There are five frame types: Problem, Question, Multiple Choice, Decision, and Copy. The Q (Question) frame will be illustrated below. The P, M, D, and C frames will be discussed briefly to point out their capabilities.* In this illustration (and in practically all PLANIT dialog) data entered by the user follow an asterik typed out by PLANIT. (In our examples, lines exceeding text width are shown indented.) All data the entered interactively via teletype.

```
          (Q) THE QUESTION FRAME
FRAME 1.φφ LABEL=*HIST
2. SQ.
*?
2. SPECIFY QUESTION.
*WHO INVENTED THE ELECTRIC LIGHT?
*
3. SA.
*A+THOMAS EDISON
```

*See Reference 7 for a detailed discussion of all frames.

```
*B ALEXANDER BELL
*
4. SAT.
*A F:THATS VERY GOOD B:3
*B R:HE INVENTED THE TELEPHONE, TRY
   AGAIN . . .
*-R:
*-C:
```

Explanation of Frame 1

First line. All frames are, automatically, serially numbered by the program. Here the LD chooses to label the frame HIST. (If he chooses not to label the frame, he will merely strike the space bar and the carriage return and pass on to Group 2. Group 1 consists merely of the frame label.)

Group 2: SQ. The LD, not sure what SQ means, types in the question mark (?). PLANIT immediately repeats the group number, 2, and elaborates. The LD then types in his question WHO INVENTED THE ELECTRIC LIGHT? PLANIT returns with an asterisk, waiting for more lines of input. PLANIT has no way of knowing when the LD is through with the question group and so returns with an asterisk for each next line. The LD ends the group by striking the space bar once and then the carriage return key. There is no theoretical limit to the number of lines that can be entered in each group. We do, however, have a practical limit of 63 lines for the entire frame.

Group 3: SA. PLANIT is asking the user to SPECIFY ANSWER. The LD now enters all the anticipated answers, tagging the first one A and the second B, etc. The plus sign (+) next to the A indicates to PLANIT that this is the correct answer. The LD then indicates the end of the group by striking a space bar and carriage return.

Group 4: SAT. User is requested to SPECIFY ACTIONS to be TAKEN, depending upon which answer the student gives.

There are four types of commands in the action group:

F: What follows is the feedback message that is to be presented to the student. If no message follows F:, PLANIT will choose one randomly from its stored list of feedback messages, according to whether the student's answer was correct or incorrect. The LD can thus enter F: by itself, knowing that the student will not get a monotonous YES or NO when he enters an answer. (Responses are usually one-word messages such as FINE, YES, CORRECT, NO, WRONG.)

R: This command instructs PLANIT to wait for another answer without printing the question again. It can be entered along with an appropriate message such as in the above example. In this case the student receives the feedback: HE INVENTED THE TELEPHONE, TRY AGAIN; and PLANIT then waits for another answer. R: by itself instructs PLANIT to print out a fixed message (WRONG, TRY AGAIN) and wait.

C: This command used alone instructs PLANIT to print out the fixed message: THE CORRECT ANSWER IS, followed by the correct answer (indicated by the plus sign in Group 3). C: can be followed only by another command—F:, R:, B: or a CALC statement. For example, C: COUNT=COUNT*1 or C:X =FACT(3). This puts PLANIT in the CALC mode. COUNT is an item in CALC, as is X. Here COUNT is to be incremented by one and X is set equal to FACT (3), i.e., the factorial of 3.

B: This instructs PLANIT to branch (B:3 means BRANCH TO FRAME 3). All frames are numbered; they can also be labeled, and a branch can be made to any numbered or labeled frame. In addition, B:LSNAM (where LSNAM is the name of another lesson) means that the lesson LSNAM will now be brought into PLANIT and executed as a part of the current lesson; the student will never know the difference. Upon completion of the lesson LSNAM, PLANIT will continue with the next frame in the original lesson. B: by itself causes PLANIT to return to the calling lesson. For example, if during lesson AA the command B:BC is encountered (where BC is a lesson name), lesson BC will be called and run until the command B:, in lesson BC, automatically returns PLANIT to lesson AA. Similarly, B:PROGM means branch to the program whose name is PROGM. When PROGM relinquishes control, execution continues with the next frame in the calling lesson.

The first user input in Group 4 is read as follows: If the student gave answer A (THOMAS EDISON), print out the message: THATS VERY GOOD and branch to Frame 3.

The second input is interpreted as follows: If the student gave answer B, print out: HE INVENTED THE TELEPHONE, TRY AGAIN . . . and wait for another answer.

The third input, prefaced by a minus sign, indicates an action to be performed if the student's answer did not correspond with either A or B—namely, for any unanticipated response, wait for another answer. (In this case, since nothing appears after the R:, PLANIT then prints out the fixed message: WRONG, TRY AGAIN, and waits for another answer.)

The LD terminates the group and thereby the frame by striking the space bar and the carriage return key.

The space bar and CR alone on any line will terminate the group. The dollar sign ($) and CR alone on any line will terminate the frame. If, for example, the LD (in the middle of Group 3) decided to end the frame and proceed to the next frame, he would

only have to enter the $ and CR alone on a line and PLANIT would respond with:

P/Q/M/D/C.

User may then, to give another illustration, build a new lesson as follows:

```
*Q
FRAME 2.φφ LABEL=*MATH
2. SQ.
*LETS SEE WHAT YOU REMEMBER ABOUT
  TEMPERATURE. USING F FOR DEGREES
*FAHRENHEIT AND C FOR DEGREES CEN-
  TIGRADE, WRITE THE FORMULA FOR
*CONVERTING FROM DEGREES FAHREN-
  HEIT TO DEGREES CENTRIGRADE.\
*HINT:  F=9*C/5+32 CONVERTS FROM
  CENTIGRADE TO FAHRENHEIT.
*
3. SA.
*φ FORMULAS ON
*A+C=(5/9)*(F—32)
*B F=9*C/5+32
*C C=(5/9)*F—32
*
4. SAT.
*A F:  B:7
*B R:YOUR ANSWER IS THE SAME AS THE
  ONE I GAVE YOU, TRY AGAIN . . .
*A F: NOW YOU'VE GOT IT. B:15
*B R:YOU'RE STILL CONVERTING FROM
  CENTIGRADE TO FAHRENHEIT, TRY
  AGAIN . . .
*BC F:NOTE THE DIFFERENCE. C: B:OUT
*—R:
*—C:
*
```

Explanation of Frame 2

First line. The LD labels this frame MATH.

Group 2. SQ. The LD enters his question. Notice the back slash (\) after CENTIGRADE on the third line. This instructs PLANIT to skip a line after printing out CENTIGRADE—to set off the following HINT. For example:

```
LETS SEE WHAT YOU REMEMBER ABOUT
  TEMPERATURE. USING F FOR DEGREES
FAHRENHEIT AND C FOR DEGREES CENTI-
  GRADE, WRITE THE FORMULA FOR
CONVERTING FROM DEGREES FAHRENHEIT
  TO DEGREES CENTIGRADE.

HINT: F=9*C/5+32 CONVERTS FROM CEN-
  TIGRADE TO FAHRENHEIT.
```

The "\" can be used anywhere in Group 2.

Group 3. SA. This group illustrates the algebraic matching ability of PLANIT (activated by the expression: φ FORMULAS ON). The student can type in any equivalent algebraic form of the correct answer and get full credit for it; e.g., C=(F—32)*(5/9), C=5*(F—32)/9, C=(5*F—160)/9, etc., are all equivalent and therefore acceptable forms. If FORMULAS is not turned on (or is deactivated by the expression: φ FORMULAS OFF), then only the exact form as typed in by the LD would be looked for in the answer. This, of course, is not true symbol manipulation; we merely employ a technique which (in part) includes performing algebra on the student's answer. The matching technique is not restricted to correct answers alone but works for all anticipated answers in Group 3.

Group 4. SAT. This group illustrates repeated use of a frame. The *first* time through this frame, if the student's answer corresponds to:

A—he receives a (randomly selected) feedback message followed by a branch to Frame 7.

B—he receives the feedback message: YOUR ANSWER IS THE SAME AS THE HINT I GAVE YOU, TRY AGAIN. . .

C—he receives: NOTE THE DIFFERENCE. THE CORRECT ANSWER IS: C=(5/9)*(F—32) . . . followed by a branch to the frame whose label is OUT.

If he gives an unanticipated answer (neither A, B, nor C), he receives the message: WRONG, TRY AGAIN.

The second time through the frame, if the student's answer corresponds to:

A—he receives: NOW YOU'VE GOT IT. PLANIT then branches to Frame 15.

B—he receives: YOU'RE STILL CONVERTING FROM CENTIGRADE TO FAHRENHEIT.
TRY AGAIN. . .

C—he receives the same message as in C above.

If no match (second unanticipated answer), he receives: THE CORRECT ANSWER IS C=(5/9)*(F—32). PLANIT will then go on to the next frame in the sequence.

If the student goes through the frame a third time, or more, and gives an answer corresponding to:

A—he receives the same feedback as for A the second time through.

B or C—he receives the same feedback as for C above.

Unanticipated answer—he receives the same feedback as for unanticipated answers the second time through.

Note: Several commands (F: C: R: B:) occurring on one line are performed in the order of their appearance from left to right.

If there is no Group 3, then all commands in Group 4 will be executed, and there will be no pause for the student's answer.

Let's go on to another example frame illustrating the PHONETIC and KEYWORD routines.

```
P/Q/M/D/C.
*Q
FRAME 3.φφ LABEL=*PRES
2.  SQ.
*WHO WAS THE FIRST PRESIDENT OF THE
  USA?
*
3.  SA.
*φ PHONETIC ON
*φ KEYWORD ON
*A+GEORGE WASHINGTON
*B ABE LINCOLN
*C+G. WASHINGTON
*
4.  SAT.
*A F: B:SOMEPLAC
*B R:HE WASN'T THE FIRST, TRY AGAIN...
  C:  COUNT=COUNT+1
*C R:SPELL HIS FIRST NAME.
*
```

Explanation of Frame 3

First line. The LD labels it PRES.

Group 3. SA. Two different correct answers are designated above by the letters A and C. This is perfectly acceptable to PLANIT. However, when the command C: is used, the correct answer printed out will always be the last one with the plus sign. This group also illustrates the use of the phonetic and keyword matchers. The phonetic matcher encodes all answers into their phonetic equivalent. The keyword function looks for the answer or answers designated by the LD anywhere in the student's response. Both phonetic and keyword are turned on, as shown in Group 3. The zero in front of PHONETIC ON and KEYWORD ON tells PLANIT to perform this function before any answers are matched. Their combined use would cause PLANIT to accept the following answer as correct: I THINK IT WAS JEORGE WASHINGTUN. KEYWORD does not accept the answer in reverse order; i.e., WASHINGTON GEORGE would not be accepted. However, if FORMULAS was turned on too, then either order would be accepted.

(M) THE MULTIPLE CHOICE FRAME

The M frame is built exactly the same as the Q frame. The only difference is that during the execution of the lesson, the choice of answers (in Group 3) is printed out. The plus sign, if any, is omitted of course.

(P) THE PROBLEM FRAME

The use of this frame is rather specialized. Three kinds of information may be inserted here by the LD:

- Probability distribution parameters for generating data in the form of random samples, e.g., means, variances, and correlation coefficient for a bivariate Gaussian (normal) distribution. The random data would actually be generated for student use during the execution, for example, of a statistics lesson (the LD can also specify headings and format for the actual printing of the data).
- Steps to the solution of a problem in which the random data will be used (the student receives one step at a time in response to his request "STEPS").
- Controls over the mathematical functions which shall (or shall not) be made available to the student when he attempts to solve a particular problem.

In addition there is another less specialized use for this frame. The LD may insert here the names of files (data bases) that can be used by PLANIT to search for answers to questions posed by students.

(C) THE COPY FRAME

The Copy frame is more of a building aid than a frame in its own right. It allows one to copy and modify any frame previously built in the same lesson and to include it in the frame presently being built.

(D) THE DECISION FRAME

As previously illustrated, all branching decisions are made as a function of what actions have taken place during execution of the frame. The Decision frame affords the LD the opportunity to consider branching decisions (and other forms of program behavior) as a function of the student's past performance, that is, as a function of results that have taken place during execution of a set of frames. Since our goal is to provide the user with a language for handling— quickly, naturally, and easily—the kinds of problems that one encounters in CHI, we devised a "language" for describing patterns of past performance. This language takes two basic forms.

The first form, called the pattern form, allows one to inquire if the student took a particular path through the material. For example, let us imagine the student went through Frame 1, answered A or C, and followed it by Frame 5 (in which he responded incorrectly), and then followed that by Frames 10 through 15, where the student was correct. An inquiry concerning whether or not the student went through that pattern exactly with no deviations can actually be written in the language pretty much as it has been stated. For example:

```
IF 1,AC 5,— 10-15,+
```

This, then, is the form of the pattern question. If the query is answered affirmatively, one can then use any combination of the three action commands:

F:, C:, and B:.

The second form permits queries about summarized student performance over a set of frames. For example,

one may want to know if the student got less than or equal to five right out of Frames 10 through 22 and Frames 30 and 33. This can also be written, almost as stated, in the following manner: IF LQ 5 RIGHT 10-22,30,33. Similarly, if these conditions are satisfied, any of the three commands can then be executed. In place of RIGHT, one can substitute WRONG, SEEN, MINUTES, USED. With USED, one can have any function like FACT (factorial), SIN, COS, etc. For example:

IF GQ 5 MINUTES 10-15 B:10
IF USED SIN 6 B:7

means: If the student spent at least five minutes on Frames 10 through 15, then branch to Frame 10. The second IF statement reads: If he used the function SIN in Frame 6, then branch to Frame 7. Also, in place of GQ one can substitute LS for less than, GR for greater than or other relationals such as LQ, NQ, EQ. Finally one can use IF statements to query the contents of items set in the calculation mode. IF IQ LS 50 B:WORK means: If the item whose name is "IQ" contains a value less than 50, branch to the frame whose label is "WORK."

All these forms can be connected as one large statement via the connectives

AND and OR, e.g.:
IF 1,AC 5,— 10-15,+
OR GQ 5 MINUTES 10-15 AND USED SIN 6
AND IQ LS 50 B:WORK

The above examples illustrate only a few uses of PLANIT. In the calculation (CALC) mode, these same capabilities can be turned to mathematical subject matter. CALC provides a powerful computing capability; arithmetic expressions can be instantly evaluated, mathematical functions can be defined, and a number of stored functions (such as the generation of pseudorandom numbers) and primitives (e.g., FACT(N) = N!) are available to teacher and student. If, for example, the student is operating in the execution (EX) mode and working on a lesson frame that requires him to perform computation, he may enter the CALC mode by typing a left arrow, instruct the machine to perform the desired operation, and receive an immediate answer, as illustrated:

	Dialogue	*Explanation*
User:	*FUNCTION A(X,Y) = X*Y	Defining of function A(X,Y) to be equal to the product of X and Y.
System:	IN	
User:	*A(5,4)	Using the function with arguments 5 and 4.
System:	20.0	
User:	*A(7,FACT(3))	Using the function with arguments 7 and the factorial of 3.
System:	42	

Current and future uses

Materials produced via PLANIT are: Instructional sequences in statistics (aimed at students of the social sciences enrolled in a first course in statistics); spelling and vocabulary (for children three to eight years); economics (for undergraduates); introduction to computer programming (for persons having some knowledge of algebra). Agencies who have used or are using PLANIT are: System Development Corporation, Southwest Regional Laboratory, University of California at Los Angeles, University of Southern California, University of California at Irvine, United States Naval Personnel Research Activity (San Diego, California), New England Educational Data System, and Lackland Air Force Base.

In view of PLANIT's interactive and evaluative capabilities, one can readily employ the system for other applications. For example, it is being currently used in the development of a computer-based vocational counseling program, and is being adapted for use in administrative planning and management. Its usefulness in this area will be greatly enriched by the CALC capability.

The PLANIT system, now operating on SDC's Q-32 time-sharing system, will soon be available for use on the IBM 360/65. In the near future we hope to make conversion to other smaller computers such as the IBM 360/30.

ACKNOWLEDGMENT

The author wishes to acknowledge the work of Dr. Charles Frye of System Development Corporation who was co-developer of the system.

REFERENCES

1 M H PERSTEIN
Grammar and lexicon for basic JOVIAL
System Development Corporation Santa Monica California Technical Memorandum TM-555/005/00 10 May 1966

2 D L BITZER J A EASLEY
PLATO: A computer-controlled teaching system
In M A Sass and W D Wilkinson Eds Symposium on computer augmentation of human reasoning Spartan Books Washington D C pp89-103 1965

3 IBM 1401 1440 or 1460 operating system computer assisted instruction
IBM Reference Publication C24-3253-1 Rev International Business Machines Corporation 1964

4 L M STOLUROW
Systems approach to instruction
University of Illinois Training Research Laboratory Technical Report no 7 July 1965

5 *A manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System* Dartmouth College Hanover New Hampshire 1 January 1965

6 J M NEWTON
ONR conference on CAI languages
Report of material presented at conference March 2-3 1966 Cambridge Massachusetts ENTELEK Incorporated Newburyport Massachusetts unnumbered technical report

7 S L FEINGOLD C H FRYE
User's guide to PLANIT
System Development Corporation Santa Monica California Tech Memo TM-3055/000/01 17 October 1966

A formal system for the specification of the syntax and translation of computer languages*

by JOHN J. DONOVAN and HENRY F. LEDGARD
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

This paper presents two basic results: the use of established methods of recursive definition to present a single method to

1. specify the syntax of computer languages (including contextsensitive requirements, such as the restrictions implied by declaration statements),
2. specify the translation of programs in one computer language into programs in another language.

The method can be used to write one specification for both the syntax of a language (e.g. a source language) and its translation into a target language (e.g. an assembler language). A syntactically legal program and its translation into the target language can then be generated using the specification. If the target language is understood, the semantics of the first language is specified.

The paper develops the method of recursive definition in conjunction with an example specifying the syntax of a limited subset of PL/I (including declaration, arithmetic, and conditional statements) and its translation into IBM System/260 assembler language.

Background

Our objective is to present a single method for expressing the syntax and translation of computer languages. The method was presented and first applied to specify only the syntax of computer languages in an earlier work by Donovan.[4]

The objective to develop methods for specifying either the syntax or the translation of computer languages is not new. In response to the demand for numerous problem-oriented computer languages to meet the needs of diverse fields, there has been considerable activity to ease the effort required to define and implement a language.

Much of this activity has led to the development of methods for specifying, at least in part, the syntax of computer languages.[9-14] The methods for specifying syntax have facilitated the description of computer languages to members of the computing field and have led to the development of syntax-directed translators.[25-28] However, most of the methods for specifying syntax have been shown to be equivalent to context-free phrase structure grammars and hence inadequate[13,23,24] for completely characterizing the syntax of computer languages. For example, some programming languages require that all statement labels in a program be different, that reference labels refer to existing statement labels, that the arithmetic type of a variable be declared, or that the dimensions of an array be declared before referring to an element within the array. In a Fortran program, for instance, the statement "GO TO 20" is not legal unless "20" occurs as a statement label in the same program. Restrictions like these cannot be specified by a context-free grammar. We consider these restrictions to be syntactic in that programs violating these restrictions are never translated, but are rejected solely on their form. Debate as to whether these restrictions are syntactic or semantic is immaterial when we wish to specify both the syntax and translation of a language, because then all these restrictions must be satisfied.

Other activity has been directed to developing table-driven compilers[27,28] and programming languages for expressing string transformations.[20,21] The table-driven compilers have generally been limited to a particular type of target language and have required excessive detail in writing the specifications to fill

*Work reported herein was supported (in part) by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

the tables for a particular source language. The string transformation languages have been limited to special types of string transformations and have not been found generally useful for translating computer languages. Approaches to the formalization of the semantics of computer languages have also been made.[16-19]

Here we present a single, formal method for specifying completely the syntax and translation of a computer language. The method is independent of both the source and target languages. The method uses an uncluttered, readable notation. The method recursively classifies sets of strings. The syntax of a computer language is characterized by specifying a set where each element is a syntactically legal program. The translation of a computer language is characterized by specifying a set of ordered pairs, where the first element of each pair is a syntactically legal program in the source language, and the second element is a corresponding program in the target language that preserves the meaning of the source language program. If the target language is understood, the semantics of the source language is specified.

The paper develops the method of recursive definition with an example specifying the syntax of a limited subset of PL/I and its translation into IBM System/360 assembler language. The power of the method of recursive definition is discussed and an ordered set of appendices is presented. The appendices present: 1) a brief summary of the notation for the method of recursive definition, 2) two programs in the subset of PL/I and their translation into System/360 assembler language, 3) a Backus-Naur Form specification of the syntax of the subset of PL/I, 4) a complete specification of the syntax of the subset using the method of recursive definition, and 5) a complete specification of the syntax of the subset and its translation into System/360 assembler language using the method of recursive definition.

Basis of formalization

The formalization for the method presented here evolved from Post's canonical systems,[1] and hence will be called *canonic systems*. Smullyan[2] used an applied variant of the canonical systems of Post in his definition of elementary formal systems. In class notes on the application of elementary formal systems to the definition of self-contained mathematical systems, Trenchard More[3] modified the definition of elementary formal systems. Elementary formal systems (now called canonic systems in recognition of the earlier work by Post) were further modified to meet the definitional needs of computer languages and applied to the definition of syntax by Donovan.[4] Canonic systems were later applied by Ledgard[5] to specify the translation of computer languages. This paper is a synthesis of the last two works.

Canonic systems, which are equivalent to elementary formal systems, can be used to specify any *recursively enumerable set*.[2] Smullyan used elementary formal systems as the basis for his entire study of formal mathematical systems and recursively enumerable sets. We use canonic systems to define two examples of recursively enumerable sets, the set of syntactically legal programs comprising a computer language, and the set of ordered pairs specifying the translation of programs in one language to programs in another language. We may feel confident that canonic systems can specify any programming language, or more generally, any algorithm or translation that a machine can perform. This confidence is a direct consequence of the works by Turing[34,35] and Kleene.[36] Here the notion of "recursive sets" (which are encompassed by canonic systems) was shown equivalent[35,36] to the notion of functions computable by a "Turing machine", and functions computable by a Turing machine were asserted[34] to comprise every function or algorithm that is intuitively computable by machine.

Canonic systems

A *canonic system* is a finite sequence of rules for recursively defining sets. The elements of the sets are strings of symbols selected from some finite alphabet. Each rule is called a canon. A canon generally has the form

$$a_1 \text{ set } A_1 \dashv a_2 \text{ set } A_2 \dashv \ldots \dashv a_n \text{ set } A_n \vdash b \text{ set } B$$

which may be interpreted informally:

> If "a_1" is a member of the set named "set A_1", and "a_2" is a member of the set named "set A_2", . . . , and "a_n" is a member of the set named "set A_n", then we can assert that "b" is a member of the set named "set B".

The "a_i" and "b" represent symbols from the finite alphabet; the "set A_i" and "set B" are the names of the sets defined.

In the remainder of this section we will elaborate on this notation. A synopsis of the notation is given in Appendix 1. and may be used as a reference throughout the text. The notation will be developed by a series of examples taken from the canonic system specifying the syntax of a subset of PL/I, called Little PL/I. This subset includes limited forms of PL/I GO TO statements, IF statements, label assignment statements, label declaration statements, and arithmetic assignment statements. The Backus-Naur

Form description of the subset is given in Appendix 3. Two example syntactically legal programs in the subset are given in Appendix 2. One of these examples is repeated here:

```
Q:      PROCEDURE;
        DECLARE LX LABEL;
L:          I = I+IA×IB−IC;
            LX = L;
            GO TO CHECK;
M:          I = I + 1;
            LX = M;
CHECK:  IF I < LIMIT
            THEN GO TO LX;
END Q;
```

We define the syntax of a language as the set of rules for specifying the strings that can be recognized by a translator and translated into some other language. The set of rules excludes strings that the translator would reject solely on their form. The syntax of Little PL/I has the following restrictions, which for all practical purposes make Little PL/I context-sensitive and therefore impossible to completely characterize in Backus-Naur Form:

1. Different declarations of the same identifier are in error, i.e.
   a. The lists of fix-pt variables, statement labels and declared label variables for a program must be mutually disjoint, and
   b. the label before PROCEDURE must not occur within the procedure block.
2. The label after END must be identical to the label before PROCEDURE.
3. All statement labels must be different.
4. The identifier in a GO TO statement must refer to an existing statement label or a declared label variable.
5. The identifier on the left hand side of the "=" in a label assignment statement must refer to a declared label variable; the identifier on the right hand side of the "=" must refer to an existing statement label or a declared label variable.

We begin our discussion of the notation for canonic systems by specifying a set named "letter". The canons for this set are as follows:

$\vdash$ A letter (1)
$\vdash$ B letter (2)
. .
. .
. .
$\vdash$ Z letter (26)

These canons may be read:

(From no premises) we can assert that the symbol "A" is a member of the set named "letter".
(From no premises) we can assert that the symbol "B" is a member of the set named "letter".
And so on.

The canons specify a set named "letter" comprising the capital letters of the English alphabet. The sign "$\vdash$" is the assertion sign. The strings "A letter", "B letter", . . . , and "Z letter" are conclusions. The capital English letters A through Z are members of the object language. The underlined, lower case characters are predicates. A predicate, here "letter", is the name of a set.

The set named "identifier" may be specified in terms of the set named "letter":

ℓ_1 letter $\vdash$ ℓ_1 identifier (27)
ℓ_1 letter $\vdash$ ℓ_2 letter $\vdash$ $\ell_1\ell_2$ identifier (28)
. .
. .
. .
ℓ_1 letter $\dashv$ ℓ_2 letter $\dashv$ . . . $\dashv$ ℓ_8 letter
$\vdash$ $\ell_1\ell_2\ell_3\ell_4\ell_5\ell_6\ell_7\ell_8$ identifier (34)

These canons may be read:

If "ℓ_1" is a member of the set named "letter", then we can assert that "ℓ_1" is a member of the set named "identifier".

If "ℓ_1" is a member of the set named "letter" and "ℓ_2" is a member of the set named "letter", then we can assert that "$\ell_1\ell_2$" is a member of the set named "identifier".
And so on.

The canons* specify a set consisting of identifiers of one to eight capital English letters. The lower case (possibly subscripted or superscripted) English letters ℓ_1 through ℓ_8 are variables that represent members of the set named "letter". The strings "ℓ_1", "$\ell_1\ell_2$" are terms. A term is a string of variables or symbols from the object alphabet (e.g. "ℓ_1", "$\ell_1\ell_2$", or "A$\ell_1\ell_2$"). The sign "$\dashv$" is the conjunction sign. The strings "ℓ_1 letter", "ℓ_2 letter", "ℓ_3 letter" are premises. The "$\dashv$" before an "$\vdash$" separates premises, all of which must be satisfied to assert the conclusion. The strings "ℓ_1 letter", "ℓ_1 identifier", "ℓ_2 letter", "$\ell_1\ell_2$ identifier" are remarks. A remark is either a premise or a conclusion.

*Actually, these canons should properly be called canon schema, which denote instances of canons.

The vocabulary used to describe a canon is summarized as follows:

ℓ_i letter $\vdash$ ℓ_i identifier

variable variable

term predicate term predicate

premise conclusion

remark remark

canon

To demonstrate the property of recursion in canons, we define a set named "list". The set named "list" is needed in the canonic system for the syntax of Little PL/I to specify the requirement that the list of all reference labels for a program must be contained in the list of all statement labels. The canons for the set named "list" are as follows:*

$\vdash \Lambda$ list (35)

i identifier $\vdash$ i, list (36)

a list $\dashv$ b list $\vdash$ ab list (37)

We can use canons (1) through (37) to derive the conclusion that the string "A,BB,A," is a member of the set named "list". Using canon (1) we can assert that "A" is a member of the set named "letter". Using canon (27) we can assert "A" is a member of the set named "identifier". Using canon (36) we can assert that "A," is a member of the set named "list". Similarly, using canons (2), (28), and (36) we can assert that "BB," is a member of the set named "list". Since the premises "A, list" and "BB, list" have been asserted, we can use the instance of canon (37)

$$A, \text{list} \dashv BB, \text{list} \vdash A,BB, \text{list}$$

to assert that "A,BB," is a member of the set named "list". Using canon (37) again (recursively) and letting "a" denote the list "A,BB," and "b" denote the list "A," we can assert that "A,BB,A," is a member of the set named "list".

The repeated use of the same predicate names in the above canons strongly suggest the use of two abbreviations:

1. If $C_1, C_2, \ldots C_n$ are conclusions with identical premises Q, the canons

   $Q \vdash C_1$
   $Q \vdash C_2$
   .
   .
   .
   $Q \vdash C_n$

   may be abbreviated

   $Q \vdash C_1 \dashv C_2 \dashv \ldots \dashv C_n$

2. If $t_1, t_2, \ldots, t_n$ are terms denoting members of the same set named "S", the remarks

   $t_1S \dashv t_2S \dashv \ldots \dashv t_nS$

   may be abbreviated

   $t_1 \dashv t_2 \dashv \ldots \dashv t_nS$

Thus the canons for "letter" may be abbreviated with abbreviation 1:

$\vdash$ A letter $\dashv$ B letter $\dashv \ldots \dashv$ Z letter

or further abbreviated with abbreviation 2:

$\vdash$ A $\dashv$ B $\dashv \ldots \dashv$ Z letter

The canons for "identifier" may be abbreviated*

$\ell_1 \dashv \ell_2 \dashv \ldots \dashv \ell_8$ letter $\vdash \ell_1 \dashv \ell_1\ell_2 \dashv \ldots$

$\dashv \ell_1\ell_2\ell_3\ell_4\ell_5\ell_6\ell_7\ell_8$ identifier

This canon may be read:

If "ℓ_1", "ℓ_2", ..., and "ℓ_8" are members of the set named "letter", then we can assert that "ℓ_1", "$\ell_1\ell_2$", ..., and "$\ell_1\ell_2\ell_3\ell_4\ell_5\ell_6\ell_7\ell_8$" are members of the set named "identifier".

We also introduce the following abbreviation:

3. In cases where many canons have some common premises, a block structure abbreviation may be used. The common premises are stated once and understood to be added to the premises of all canons within the scope of the common block.

This abbreviation greatly eased the writing of the specification for the syntax of the simulation language GPSS.[4] There was no need to incorporate this abbreviation into the canonic system of Little PL/I. An abbreviation of two or more canons is formally called an edict. However, we will usually refer to edicts as canons.

Thus far we have specified only sets of 1-tuples, i.e., predicates of degree 1. For a complete specification of Little PL/I, we will have to specify sets of n-tuples for n greater than 1, i.e., predicates of degree n, n>1. We use the notation

$<a_1 <a_2 \ldots <a_n>$, where $a_1, a_2, \ldots, a_n$ are terms,

to denote an ordered n-tuple. The sign "$<$" separates elements of an ordered n-tuple. For the requirement that the list of reference labels for a Little PL/I program be contained in the list of statement labels, we specify a set named "in". The predicate "in" names the set of all ordered pairs such that the second ele-

*The symbol "Λ" denotes the null string.

*To obtain the given abbreviation for "identifier", we must first change the premises of each of the eight unabbreviated canons to "ℓ_1 letter $\dashv \ell_2$ letter $\dashv \ldots \dashv \ell_8$ letter". This change does not effect the validity of the canons.

ment is a list of identifiers and the first element is a list of identifiers contained in the second list.

The set named "in" is defined:

aϕbϕc list ⊢ <b$_{<}$abc> in (38)

<a$_{<}\ell$> in $\dashv$ <b$_{<}\ell$> in ⊢ <ab$_{<}\ell$> in (39)

To suggest more strongly the relationship among the elements of an n-tuple, we allow premises and conclusions of the form

$$<a_1<a_2< \ldots <a_n> \text{text1 text2} \ldots \text{textn} -1$$

to be written

$$a_1\text{text1 } a_2 \text{ text2} \ldots \text{textn} -1 \quad a_n$$

Thus the canons for "in" become

aϕbϕc list ⊢ b in abc (38)

a in ℓ $\dashv$ b in ℓ ⊢ ab in ℓ (39)

These canons may informally be read:

If "a", "b", and "c" are lists, then we can assert that the list "b" is contained in the list "abc".

If the list "a" is contained in the list "ℓ" and the list "b" is contained in the list "ℓ", then we can assert that the list "ab" is contained in the list "ℓ".

The following conclusions can be derived from canons (1) through (39):

A, in A,B, A,A, in A, R,S, in S,T,R,

ALPHA, in ALPHA,BETA,

Finally, we extend the use of the symbol "ϕ" to abbreviate the writing of n-tuples by allowing remarks of the form

$$a_1\ \text{text1}\ a_2\ \text{text2} \ldots \text{textn} -1 \quad a_n$$

$$a'_1\ \text{text1}\ a'_2\ \text{text2} \ldots \text{textn} -1\ a'_n$$

to be abbreviated

$$a_1\phi a'_1\ \text{text1}\ a_2\phi a'_2\ \text{text2} \ldots \text{textn} -1 \quad a_n\phi a'_n$$

Thus canon (39)

a in ℓ $\dashv$ b in ℓ ⊢ ab in ℓ (39)

may be abbreviated

aϕb in $\ell\phi\ell$ ⊢ ab in ℓ (39)

Canonic system specification of syntax

This section is concerned with the motivation and development of a canonic system for the syntax of Little PL/I, including context-sensitive requirements. Our approach will be to specify eventually a set of 1-tuples named "PL/I program". Each member of this set will be a syntactically legal Little PL/I program. The entire canonic system specification of this set is given in Appendix 4. The numbers of the BNF productions of Appendix 3. and the canons of Appendix 4. correspond in that productions and canons with corresponding numbers specify corresponding syntactic constructions.

The sets of letters and digits are specified by canons 1. and 2. of Appendix 4. An identifier in Little PL/I is a string of one to eight letters. Canon 3. specifies the set of identifiers. Similarly, the set of unsigned integers, whose members are strings of one to ten digits, is specified by canon 4. Little PL/I has only fixed-point arithmetic variables. Canons 5.1 and 5.2 specify the set of fixed-point variables. Canons 6. and 7.1 specify the set of labels and label variables.

To specify the restriction that a reference label in a GO TO statement be contained in the list of statement labels or that all label variables for a program be contained in the list of declared label variables, we define the sets named "list" and "in" (canons 7.2 through 7.6). The canons for "list" specify a set of lists, where each list is a sequence of identifiers separated by commas. The canons for "in" specify a set of *ordered pairs* of lists, where each identifier in the first list is contained in the second list.

To specify the restrictions that the sets of fixed-point variables, statement labels, declared label variables, and the procedure label for a program must be mutually disjoint, we define the set of ordered pairs named "disjoint" (canons 7.10 through 7.13). The first element of each ordered pair is a list of identifiers; the second element of each pair is a list of identifiers, none of which appears in the first list. For example, the ordered pair "<A,B,C,$_{<}$D,E,F,>" is a member of this set. To define the set named "disjoint", we first define a set named "differ" (canons 7.7 through 7.9). Members of the set named "differ" are ordered pairs, in which the first element is an identifier and the second element is a different identifier.

Canons 8.1 through 18.4 specify the constructions for Little PL/I primaries, GO TO statements, relational operators, boolean expressions, IF statements, label assignment statements, arithmetic assignment statements, and label DECLARE statements. For example, with every GO TO statement we keep track of its reference label (to check later if it is in the list of statement labels) or keep track of its label variable (to check if it is in the list of declared label variables). Thus we define the canon for a GO TO statement:

ℓ label ⊢ GO TO ℓ; goto stm with ref label ℓ, label var Λ (9.1)

This canon specifies a set of 3-tuples named "goto stm with ref label-label var". The first element of a 3-tuple is a GO TO statement, the second element the reference label for the GO TO statement, the third element the label variable for the GO TO statement. The canon has the following instance:

A label ⊢ GO TO A; goto stm with ref label A, label var Λ

or

A label ⊢ <GO TO A;<A,<Λ> goto stm with ref label-label var

Likewise, with every arithmetic assignment statement we keep track of the list of its fixed-point variables (to check later if the list of fixed-point variables for a program is disjoint from the lists of statement labels, declared label variables, and procedure label for the program). Hence the canons for an arithmetic expression and an arithmetic assignment statement specify sets of ordered pairs, in which the second element of each pair is the list of associated fixed-point variables. In the same manner, we define a set of ordered 4-tuples to specify an IF statement, a set of ordered 3-tuples for a label assignment statement, a set of ordered pairs for a DECLARE statement.

Canons 19.1 through 19.4 specify a statement sequence consisting of a single (possibly labeled*) statement. Canon 20.1

sϕs′ stm seq with stm labels $\ell_s \phi \ell'_s$ ref labels $\ell_r \phi \ell'_r$
label vars $\ell_v \phi \ell'_v$ decl label vars $\ell_{vd} \phi \ell'_{vd}$
fix-pt vars vϕv′ $\nmid$ ℓ_s disjoint ℓ'_s (20.1)
⊢ ss′ stm seq with stm labels $\ell_s \ell'_s$ ref labels $\ell_r \ell'_r$
label vars $\ell_v \ell'_v$ decl label vars $\ell_{vd} \ell'_{vd}$ fix-pt vars vv′

specifies a statement sequence of two or more statements. This canon may informally be read

If "s" is a stm seq with stm labels "ℓ_s", ref labels "ℓ_r", label vars "ℓ_v", declared label vars "ℓ_{vd}", and fix-pt vars "v",
and "s" is a stm seq with stm labels "ℓ_s", ref labels "ℓ'_r", label vars "ℓ'_v", declared label vars "ℓ_{vd}'", and fix-pt vars "v'",
and the list of stm labels "ℓ_s" is disjoint from the list of stm labels "ℓ_s'",
then we can assert that "ss'" is a stm seq with stm labels "$\ell_s \ell'_s$", ref labels "$\ell_r \ell'_r$", label vars "$\ell_v \ell'_v$", declared label vars "$\ell_{vd} \ell'_{vd}$", and fix-pt vars "vv'".

*The element "stm labels" for a labeled DECLARE statement is given as "Λ" since statement labels for DECLARE statements are ignored in PL/I.

The premise "ℓ_s disjoint ℓ'_s" insures that the statement labels for each statement sequence are different.

Canon 20.2

s stm seq with stm labels ℓ_s ref labels ℓ_r label vars ℓ_v
decl label vars ℓ_{vd} fix-pt vars v $\nmid$ v disjoint $\ell_s \ell_{vd}$
$\nmid$ ℓ_s disjoint ℓ_{vd} $\nmid \ell_r$ in ℓ_s $\nmid$ ℓ_v in ℓ_{vd} (20.1)
⊢ s legal stm seq with stm labels ℓ_s vars ℓ_{vd}v

specifies a syntactically legal statement sequence. The premises "v disjoint $\ell_s \ell_{vd}$" and "ℓ_s disjoint ℓ_{vd}" insure that the lists of declared label variables, fixed-point variables, and statement labels for a Little PL/I program are mutually disjoint. The premises "ℓ_r in ℓ_s" and "ℓ_v in ℓ_{vd}" insure that all reference labels refer to existing statement labels and that the label variables used in the label assignment and GO TO statements are declared.

Finally, canon 21.

ℓ label $\nmid$ s legal stm seq with stm labels ℓ_s vars v
$\nmid$ ℓ, disjoint ℓ_sv ⊢ ℓ: PROCEDURE; s END ℓ;
PL/I program.

specifies a syntactically legal PL/I program. The premise "ℓ, disjoint ℓ_sv" insures that the procedure label "ℓ" is not used within the statement sequence for the program. The set named "PL/I program" is our desired set, the set of 1-tuples such that each member of the set is a syntactically legal Little PL/I program.

We thus state our first basic result:

1. The language of canonic systems can be used to specify (exactly) the syntax of a computer language.

Canonic system specification of translation

We know that theoretically, at least, the translation of programs from one computer language to programs in another computer language is a process that *can* be expressed by a canonic system. We can view every translator as specifying a function from one set of strings to another. Since the translation of programs is a task performed on computers, we can assert that the function specifying the translation is recursively enumerable and hence can be specified by a canonic system. The ordered pairs defining the function comprise the recursively enumerable set. The first element of each ordered pair is a program in one language, the second element its translation into the target language.

We use this last fact to motivate our development. We will develop the specification for the translation of a language by

a. modifying the specification of the syntax of the language to distinguish further the *semantically*

different strings (e.g., the set of arithmetic operators +, −, *, and / will be split into two sets, one for the addition operators + and − and one for the multiplication operators * and /), and

b. appending to each n-tuple specifying a translatable string one more element specifying a corresponding string in some other language that preserves the meaning of the first string.

Instead of eventually specifying a set of 1-tuples comprising the set of syntactically legal programs (as we did in the previous section), we will eventually specify a set of ordered pairs. The first element in each pair will be a syntactically legal program, the second element its translation into the target language.

As in the previous section, we will illustrate our approach by example. The syntactic specification of Little PL/I will be modified to specify not only the syntax of Little PL/I but also its translation into IBM System/360 assembler language. A complete specification of the syntax and translation of Little PL/I is given in Appendix 5. Two example syntactically legal programs and their translations specified by Appendix 5. are given in Appendix 2. As in the example translations, the canons of Appendix 5. specify comments entries with the assembler statements so that (hopefully) the reader will not have to be familiar with IBM System/360 assembler language to understand the translation.

In the succeeding paragraphs of this section we present a discussion of the canons of Appendix 5. The reader may wish to omit this discussion. The techniques used in forming the canons of Appendix 5. may be grasped by comparing the correspondingly numbered canons of Appendices 4. and 5. For example, canons 9.1 and 9.2 of Appendix 4. give the canons specifying the syntax of a GO TO statement, and canons 9.1′ and 9.2′ of Appendix 5. give the canons specifying the syntax of a GO TO statement and its translation into assembler language. Following this section we indicate further applications of canonic systems and conclude with a discussion of the use of canonic systems as a method of definition.

Consider first canon 9.1 of Appendix 4:

$$\ell \text{ label} \vdash \text{GO TO } \ell\text{; goto stm with ref label } \ell, \quad \text{label var } \Lambda \tag{9.1}$$

The assembler language statement for a GO TO statement such as "GO TO A;", where A is a reference label, is simply

```
B  A      *BRANCH TO A
```

where "B" is the operation code for an assembler branch statement, "A" is the symbolic address* of the first assembler statement for the referenced Little PL/I statement, and "*BRANCH TO A" is a comments entry. We may specify this translation by modifying canon 9.1:

$$\ell \text{ label} \vdash \text{GO TO } \ell\text{; goto stm with ref label } \ell, \quad \text{label var} \Lambda \text{assembler stms} \tag{9.1′}$$

```
B  ℓ      *BRANCH TO ℓ
```

The modified predicate "goto stm with ref label-label var-assembler stms" names a set of 4-tuples, where the elements of a 4-tuple are

1. a GO TO statement
2. the reference label for the GO TO statement
3. the label variable for the GO TO statement
4. the assembler statement for the GO TO statement.

The instance of canon 9.1′ for the GO TO statement "GO TO A;" is simply

$$A \text{ label} \vdash \text{GO TO A; goto stm with ref label A}, \quad \text{label var} \Lambda \text{assembler stms}$$

```
B  A      *BRANCH TO A
```

Similarly, canon 9.2 of Appendix 4., the canon for a GO TO statement with a label variable, is modified to specify its translation in canon 9.2′.

The assembler language statements for an IF statement, such as

IF I<5 THEN <non-DECLARE statement>,

might be as follows:

```
L    1,I       *LOAD I
C    1,=F'5'   *COMPARE WITH =F'5'
BNL  Z         *BRANCH IF NOT LOW TO Z
•    (translation for non-DECLARE statement
•    following the boolean expression)
•
Z EQU *        *SYMBOLIC ADDRESS OF Z
```

For these assembler statements we must know:

a. The operand entries (I and =F'5') for the primaries (I and 5) of the boolean expression
b. the branch operation code (BNL) for the relational operator (<)
c. the symbolic address (Z) of the assembler statement to which control should be passed if the boolean expression is not true.

These requirements necessitate that we

*We assume that identifiers occurring in a Little PL/I statement will not be changed upon translation into assembler statements so that the same identifiers may be used.

a. modify the canon for primaries to specify the operand entry for each primary
b. modify the canon for relational operators to specify the operation code for each relational operator
c. add a premise to canon 11. to specify a new label
d. modify the conclusion of canon 11. to carry the new label for use (canon 12.') in specifying the EQU assembler statement needed to terminate the assembler statements for the IF statement.

Canons 10.', 11.', and 12.', in conjunction with cannons 8.1' and 8.2', specify the translation of an IF statement according to this format. The premise "ℓ, disjoint ℓ_a" in canon 12.' insures that the new branch label differs from any that occur in the translation of the non-DECLARE statement "s". The element "assembler labels" in the predicate for an IF statement is specified so that the label needed to speciby the proper transfer of control for a false boolean expression may later be specified to be different from the other assembler labels and PL/I identifiers.

Canons 13.1 and 13.2 of Appendix 4., the canons for a label assignment statement, are easily modified to specify their translation in canons 13.1' and 13.2'.

We next consider canons 14., 15., and 16., of Appendix 4., the canons for an arithmetic expression:

⊢ +ϕ−ϕ*ϕ/ arith op (14)

p primary with fix-pt vars v ⊢ p arith exp (15)

with fix-pt vars v aϕa' arith exp with fix-pt vars vϕv' ⊣ o arith op⊢ aoa' arith exp with fix-pt vars vv' (16)

We cannot immediately add to each of these canons the elements specifying the correct assembler language statements. In the evaluation of arithmetic expressions, the operations of multiplication and division are carried out before the operations of addition and subtraction. We may specify this requirement for a left to right evaluation of arithmetic expressions by:

a. separating the arithmetic operators (+,−,*, and /) into two classes, one for the addition operators, + and −, and one for the multiplication operators, * and /, (canons 14.1' and 14.2'),
b. defining a new construct "term" that consists of a sequence of two or more primaries separated by multiplication operators, * and /,
c. re-defining an arithmetic expression as a sequence of one or more primaries or terms separated by addition operators, + and −.
d. specifying that the left-most primary or term in an arithmetic expression be evaluated first, in machine register 1 (canons 15.1' through 15.4' for result in register 1),
e. specifying that succeeding primaries in an arithmetic expression be directly added or subtracted to machine register 1 (canon 15.5'),
f. specifying that succeeding terms in an arithmetic expression be formed first in machine register 3 (canons 15.2' and 15.3' for result in register 3) before being added or subtracted to machine register 1 (canon 15.6').

For example, canons 15.1' and 15.5' specify "I+1" as a legal arithmetic expression with assembler statements:

```
L    1,I          *LOAD I
A    1,=F'1'      *ADD=F'1'
```

Canons 15.1', 15.2', 15.3' and 15.6' specify "A+B*C*D" as a legal arithmetic expression with assembler statements:

```
L    1,A    *LOAD  A                (canon  15.1')
L    3,B    *LOAD  B                (canon  15.2')
M    2,C    *MULTIPLY BY C
M    3-1,D  *MULTIPLY  BY  D        (canon  15.3')
AR   1,3    *ADD   REGISTERS        (canon  15.6')
```

The remaining canons of Appendix 5. readily follow. Canon 16'., specifies the assembler store instruction appended to the assembler statements for an arithmetic expression. Canon 17'., is identical to canon 17. of Appendix 4. Canons 18.1' through 20.2', the canons for a non-DECLARE statement and a statement sequence, are similar to those of Appendix 4., with the added predicate elements needed to carry the associated assembler labels and assembler statements. The premise "ℓ_a disjoint ℓ'_a" added to canon 20.1' insures that the assembler labels differ from each other. The premise "ℓ_a disjoint ℓ_s ℓ_{vd}v" added to canon 20.2' insures that the assembler labels differ from the other identifiers in a Little PL/I program.

Canon 21.1' and 21.2' are new. These two canons specify the assembler data storage statements for a set of variables. For example, if "I" were the only variable in a PL/I program, the following instances of canons 21.1' and 21.2' would specify the storage statements for "I":

⊢ Λset of vars with assembler data stmsΛ
Λset of vars with assembler data stmsΛ ⊣ I identifier
⊣ I, disjointΛ ⊢ I, set of vars with assembler data stms

```
I    DS     F          *STORAGE FOR I
```

Finally, canon 21., of Appendix 4., the canon for a syntactically legal little PL/I program, is modified to give canon 21.3'

ℓlabel | s legal stm seq with stm labels ℓ_s
vars v assembler stms s_a $\notin$ ℓ, disjoint ℓ_sv $\notin$
v' set of vars with assembler data stms s_d $\notin$ v in v'
$\vdash$ ℓ: PROCEDURE; s END ℓ; PL/I program with translation (21.3')

```
*
* ASSEMBLER LANGUAGE PROGRAM FOR ℓ
*
  BALR 15,0 *SET REGISTER 15 AS BASE
                              REGISTER
  USING*,15 INFORM ASSEMBLER THAT
                              R15 IS BASE REG
  sa-
  SVC  0    *RETURN TO SUPERVISOR
*
  sd
  END       *TERMINATE ASSEMBLY
```

The added premise "v' set of vars with assembler data stms s_d" specifies the assembler data storage statements for the variables of the legal statement sequence.

The set named "PL/I program with translation" is our desired set. This predicate names the set of all ordered pairs such that the first element of each ordered pair is a syntactically legal Little PL/I program and the second element is its translation into assembler language.

We thus state our second basic result:

2. The language of canonic systems can be used to specify the translation of programs in one computer language into programs of another computer language.

If the predicates needed to specify the syntax of a source language are properly modified, the language of canonic systems can be used to write one specification for both the syntax of a source language and its translation into a target language.

Related applications

An immediate use of canonic systems is in the development of a generalized translator, i.e., a translator that is independent of both source and target languages. We have used canonic systems to define a set by specifying canons for **generating** its members. To use a canonic system specification of the syntax and translation of a language as a data base for a generalized translator, an algorithm to **recognize** source language strings specified by a canonic system and construct their associated translation is needed. No algorithm for recognizing and translating strings specified by a canonic system is presented in this paper. However, work on the application of canonic systems to a generalized translator has been initiated and is more fully described elsewhere.[6,7,8]

Canonic system may be used to obtain structural information[37] (e.g., syntactic trees) of programs. The series of canons used in generating a string provides structural information about the generated string. This series of canons constitute a "derivation" of the generated string. Only an intuitive notion of a derivation has been indicated here. However, using the concept of higher level canonic systems, the rules for constructing a derivation can be formalized, and, in turn, the rules for constructing a structural description of a derived string made explicit. Higher level canonic systems, and their application to provide structural descriptions, are also described elsewhere.[6,7]

DISCUSSION

Many features of PL/I were omitted in choosing the subset Little PL/I. We have ignored the specification of allowable spacing of Little PL/I programs. Spacing, as well as card format, has been specified for another computer language [4] using canonic systems. Some of the other omitted features (e.g., nested arithmetic expressions) would have required only minor modifications to the specifications of Appendices 4. and 5. The use of identifiers containing up to thirty-six characters would have required that the identifiers in a program be changed upon translation to assembler language, where identifiers of only eight or less characters are allowed. The use of both fixed and floating point variables was initially considered, but was omitted because many additional canons and several additional elements were needed to specify the translation of arithmetic expressions. The inclusion of these features would not have added any new ideas to our development. The large number of other PL/I features, such as the assortment of data types, block structure, procedure references and definitions, and input/output were disregarded. It is certainly within the capacity of canonic systems to specify both the syntax and translation of these features.

However, an equally important issue is whether canonic systems provides a natural and concise specification of these features. A canonic system for the complete syntax of only one language, the simula the complete syntax of one language, the simulation language GPSS,[4] has been written.

It is clear that a complete specification of the syntax of PL/I and its translation to assembler language might be unduly large. We feel that this largeness is due to three principal factors. First, the PL/I and System/360 assembler languages are complicated, and the degree of the canonic system predicates grows as the complexity of the source or target language grows. Second, on some features PL/I and IBM System/360 assembler language are poorly matched.

For example, many modifications to the canons of Appendix 5. would have been required to specify the assembler statements for the evaluation of arithmetic expressions containing both fixed and floating point variables. For any source and target languages, if a canonic system of the syntax of the source language and its translation to the target language were written before finally defining the languages, it is likely that we could better match the languages. The unwieldiness of portions of the canonic system would clearly indicate where the languages were ill-matched. Third, we feel that a part of the unwieldiness is due to the shortcomings of assembler language as a general-purpose target language. We feel strongly that if a better set of language primitives were devised, the specification of the translation process, using the new language as a target language, would be greatly eased.

It is important to develop languages whose descriptions are concise. The Backus-Naur Form specification of Appendix 3., and the associated five English sentences given in text describing the context-sensitive requirements, provide a very concise description of the syntax of Little PL/I. In their present form canonic systems will not replace Backus-Naur Form and the English language for describing programming languages to people. Present research has yielded a notational convention that will eliminate much of the detail of Appendices 4. and 5. We hope that this modification to the notation for canonic systems will make canonic systems as suitable as Backus-Naur Form for describing programming languages to people. However, in their present form canonic systems, although yielding larger specifications, provide complete descriptions in a precise language that a machine can be instructed to understand with present techniques.[8] Moreover, the language of canonic systems, like Backus-Naur Form, is readable.

We wish to point out two additional features of the canonic systems of Appendices 4. and 5. First, barring any inadvertent errors, the canonic systems describe a set of PL/I programs and assembler language programs that will run on a computer when translated by a PL/I compiler or System/360 assembler Second, the specification of the comments entries in the assembler language statements was provided not only to aid the reader. The comments are meaningful context-sensitive strings in the English language. The specification of these strings was handled as easily as the specification of the strings in assembler language. The specification of the strings in the English language illustrates the use of canonic systems to specify the entire operation of a translator, including the specification of meaningful comments. Moreover, it suggests the capacity of canonic systems to handle communication and translation in languages *other than* computer programming languages. We have not explored this enticing area.

Canonic systems are applicable to the definition of an "abstract" syntax[15] of a language. An abstract syntax of a language is a description of the syntax of a language that is independent of any actual representation of the symbolic expressions in the language. By 1) omitting the canons specifying only symbols in the object language (e.g., the canons for "letter" and "digit"), and 2) using the application of the abstract syntactic functions in place of elements specifying strings in object language (e.g., writing the canon for a Little PL/I GO TO statement as "ℓ label ϕ GO (ℓ) goto stm with ref label ℓ, label var Λ", where "GO" is a function to be specified in defining a "concrete" syntax of Little PL/I), a canonic system could be developed to provide an exact specification of an "abstract" syntax of a language.

As mentioned earlier, the results of this paper apply to any recursively emunerable set. Any function or relation that is recursively enumerable can be specified by a canonic system. Canonic systems can be used to express language and string transformations of a much more different nature than given here. The facility with which comments entries were specified suggests many uses, for example, in handling interterminal computer communication. We do not know to what extent canonic systems can practically be used to express more varied algorithms than those for string transformations.

We have used canonic systems to present a single method for specifying the syntax and translation of computer languages. To ease further the specification of the translation of computer languages it would be desirable to use a target language other than an assembler language. This new target language would have a set of primitives in which the constructions in a large class of languages could readily be expressed. This target language has not been developed.

REFERENCES

The work presented in this paper has evolved from the following works:

1 EMIL L POST
Formal reductions of the general combinatorial decision problem
Am J Math 65 pp 197-217 1964 This work presents a formal system named canonical systems and demonstrates that every system in canonical form can formally be reduced to a system in normal form.

2 RAYMOND M SMULLYAN
Theory of formal systems
Princeton University Press Princeton New Jersey 1961 This work describes a variant (elementary. formal systems) of the canonical systems of Post as the basic formalization

for a study of formal mathematical systems and recursively enumerable sets.

3 TRENCHARD MORE
Class notes for course EAS 313b applied discrete mathematics
Yale University New Haven Conn spring 1965 In this work the definition of elementary formal systems were modified to parallel More's definition of propositional complexes ref. 32 and applied to the study of various mathematical systems.

4 JOHN J DONOVAN
Investigations in Simulation and Simulation Languages
Ph.D. dissertation Yale University New Haven Conn fall 1966. Elementary formal systems were further modified (now called canonic systems) in recognition of earlier work of Post to meet the definitional needs of programming languages and applied to the definition of the syntax of computer languages.

5 HENRY F LEDGARD
A scheme for the translation of computer languages
Ph.D. thesis proposal MIT Cambridge Mass July 1967 Here canonic systems were applied to specify the translation of computer languages.

The following references 1) describe canonic systems and their use in specifying syntsx and translation, and 2) present higher level canonic systems, their use in formalizing the notion of a derivation of a program string, and a discussion of the application of canonic systems to a generalized translator for computer languages.

6 JOHN J DONOVAN and HENRY F LEDGARD
Canonic systems and their application to programming languages
Project MAC memorandum MAC-M-347 MIT Cambridge Mass April 1967

7 JOHN J DONOVAN and HENRY F LEDGARD
Canonic systems and their application to computer languages
Submitted paper August 1967

The following references 1) describe canonic systems and their use in specifying syntax and translation, and 2) present higher level strings specified by a canonic system and generate their translation.

8 JOSEPH W ALSOP
A canonic translator
B. S. Thesis MIT Cambridge Mass June 1967

The following references describe other formalizations used to characterize computer languages.

9 J W BACKUS
The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference
Proc Inter'l conf information processing UNESCO Paris pp 125-132 June 1959

10 D E KNUTH
Backus-normal form vs Backus-Naur form
Comm ACM vol 7 p 735 December 1964

11 W H BURKHARDT
Metalanguage and Syntax specification
Comm ACM vol 8 pp 304-305 May 1965

12 K E IVERSON
A method of syntax specification
Comm ACM vol 7 pp 588-589 October 1964

13 R W FLOYD
The syntax of programming languages—a survey
IEEE Trans of electronic computers pp 346-353 August 1964

14 T B STEEL Editor
Formal language description languages for computer programming
North-Holland Publishing Co Amsterdam The Netherlands 1966

15 JOHN McCARTHY
Towards a mathematical science of computation
Proc of IFIP congress Munich 1962 North-Holland Publishing Co Amsterdam pp 21-23 1963

The following references describe formalizations used to characterize the semantics of computer languages.

16 PETER LANDIN
The λ-calculus approach
Advances in programming and non-numerical computation pergamon press 1966

17 A VAN WIJN-AARDEN
Recursive definition of syntax and semantics
IFIP working conf Baden September 1964

18 J A FELDMAN
A formal semantics for computer languages and its application in a compilier-complier
Comm ACM vol 9 pp 3-9 January 1966

19 N WIRTH and H WEBER
EULER a generalization of ALGOL and its formal definition PART I
Comm ACM vol 9 pp 13-23 January 1966 PART II Comm ACM vol 9 pp 89-99 February 1966

The following papers describe programming languages for expressing string transformations.

20 D J FARBER R E GRISWOLD and I P POLONSKY
SNOBOL a string manipulation language
Journal of the ACM vol 11 no 2 pp 21-30 January

21 CALVIN N MOOERS
TRAC a procedure-describing language for the reactive typewriter
Comm of the ACM vol 9 no 3 pp 215-219 March 1966

The following paper describes a computer program for expressing transformations on natural language input strings.

22 JOSEPH WEIZENBAUM
ELIZA—A computer program for the study of natural language communication between man and machine
Comm of the ACM vol 9 no 1 pp 36-45 January 1966

The following references and ref 13 point out inadequacies of past specifications in specifying the syntax of programming languages.

23 P GILBERT
On the syntax of algorithmic languages
J ACM vol 13 pp 90-107 January 1966

24 A C DiFORINO
Some remarks on the syntax of symbolic programming languages
Comm ACM vol 6 pp 456-460 August 1963

The following references and (ref. 18) describe generalized translators that have been implemented as table-driven compilers.

25 E T IRONS
A syntax directed complier for ALGOL 60
Comm ACM vol 4 pp 51-55 January 1961

26 R A BROOKER and D MORRIS
A general translation program for Phrase-structure languages
J ACM vol 9 pp 1-10 January 1962

27 T E CHEATAM JR and K SATTLEY
Syntax-directed compiling
Proc spring joint computer conf Spartan Books Baltimore Md vol 25 pp 31-57 1964

28 C L LIU G D CHANG and R E MARKS
The design and implementation of a table driven compiler system
AFIPS vol 30 1967 Spring Joint Computer conference Spartan Books Washington 1967 A detailed discussion of this translator is given in a project MAC technical report MAC-TR-42

The following references describe to an extent the syntax of PL/I.

29 H CHLADEK V KUDIELKA and E NUEHOLD
Syntax of PL/I
IBM Technical report TR 25.058 Vienna Laboratory September 1965

30 ____, formal definition of PL/I IBM Technical Report TR 25.071 by PL/I definition group of the Vienna Laboratory December 1966

Definitions of the logical terminology used in this paper may be found in the following works.

31 PAUL C ROSENBLOOM
The elements of mathematical logic
Dover publications Inc 1780 Broadway New York 19 N Y 1950

32 TRENCHARD MORE
Relat ons between simplicational calculi
PhD dissertat on MIT Cambridge Mass May 1962

33 ALONZO CHURCH
Introduction to mathematical logic
Princeton New Jersey 1956 vol 1

34 A M TURING
On computable numbers with an application to the entscheidungs problem
proc London Math soc vol 42 pp 230 265 1936

35 A M TURING
Computability and lambda definability
J symb logic vol 2 pp 153-163 1937

36 STEPHEN C KLEENE
Lamba-definability and recursiveness
Duke math j vol 2 pp 340 353 1936

37 NOAM CHOMSKY
On certain formal properties of grammars
in readings of mathematical psychology Luce R D et al John Wiley and Sons Inc New York N Y vol 2 pp 125-155 1965

The following manuals were used in developing the example subset of Little PL/I and defining its translation to IBM System/360 assembler language.

38 ____ *IBM System/360 operating system PL/I language specification*
IBM systems reference library form C28-6571-3)

39____, *A programmer's introduction to the IBM system/360 architecture instructions and assembler language*
Student text form C20-1646-1

40 ____, *IBM system/360 operating system assembler language*
IBM system reference library form C28-6514-4

Appendix 1 SUMMARY OF NOTATION FOR CANONIC SYSTEMS

To provide the reader with a concise reference, the canonic systems notation used in this paper will be presented by a series of short examples.

Basic Constructions:

Canon(s)	Interpretation	New Construction(s)
1. ⊢ A letter	(From no premises, we can conclude that) "A" is a member of the set named "letter".	"⊢" is the assertion sign; "A letter" is a conclusion; "letter" is the name of a set and "A" is an element in the set. The name of the set, here "letter", is called a predicate.
2. ⊢ B letter	"B" is a member of the set named "letter".	____
3. ⊢ C letter	"C" is a member of the set named "letter".	____
4. *l* letter ⊢ *l* identifier	If "*l*" is a member of the set named "letter", then we can conclude that "*l*" is a member of the set named "identifier".	"*l* letter" is a premise; The small letter "*l*" is a variable denoting any member of the set named "letter".
5. i identifier ¢ j identifier ⊢ ij identifier	If "i" is a member of the set named "identifier" and if "j" is a member of the set named "identifier", then "ij" is a member of the set named "identifier".	"¢" is the conjunction sign; "¢" before the "⊢" separates premises, all of which must be satisfied to assert the conclusion.
6. ⊢ $<A_<B>$ differ	The ordered pair "$<A_<B>$" is a member of the set named "differ". The set named "differ" consists of ordered pairs whose elements are different.	The notation "$<A_<B>$" denotes an ordered pair. In general, the notation $<a_1<a_2<\cdots<a_n>$ denotes an ordered n-tuple.
7. ⊢ $<A_<C>$ differ	The ordered pair "$<A_<C>$" is a member of the set named "differ".	____

Abbreviations and alternate notations:

Canon(s)	Interpretation	New Construction(s)
8. ⊢ A letter ¢ B letter	"A" is a member of the set named "letter" and "B" is a member of the set named "letter" (same as canons 1. and 2.).	"¢" after the "⊢" separates conclusions, all of which can be asserted if the premises (here none) are satisfied.
9. ⊢ A ∤ B letter	"A" and "B" are members of the set named "letter" (same as canon 8.).	"∤" is used to separate two or more elements that are members of the same set.
10. ⊢ A differ B	The ordered pair "$<A_<B>$" is a member of the set named "differ" (same as canon 6.).	Alternate notation for "$<A_<B>$ differ". In general, the notation "a_1 text1 a_2 text2 ... textn-1 a_n" is an alternate form for "$<a_1<a_2<\cdots<a_n>$ text1 text2 ... textn-1", where "$<a_1<a_2<\cdots<a_n>$" is an n-tuple and "text1 text2 ... textn-1" is the name of the set in which the n-tuple is a member.
11. ⊢ A ∤ A differ B ∤ C	The ordered pairs "$<A_<B>$" and "$<A_<C>$" are members of the set named "differ" (same as canons 6. and 7.).	"∤" is used to separate the elements of two or more ordered pairs that are members of the same set. This notation is extended to handle n-tuples.

Appendix 2 TWO SYNTACTICALLY LEGAL PROGRAMS IN LITTLE PL/I AND THEIR TRANSLATION INTO IBM SYSTEM/360 ASSEMBLER LANGUAGE

Legal Program P

```
P:      PROCEDURE;
        A:      I = I + 1;
                IF I < 5 THEN GO TO A;
END P;
```

Translation for P

```
*
*  ASSEMBLER LANGUAGE PROGRAM FOR P
*
               BALR 15,0        *SET REGISTER 15 AS BASE REGISTER
P              USING *,15       *INFORM ASSEMBLER THAT R15 IS BASE REG
A              EQU  *           *SYMBOLIC ADDRESS FOR PL/I LABEL A
               L    1,I         *LOAD I
               A    1,=F'1'     *ADD = F'1'
               ST   1,I         *STORE RESULT IN I
               L    1,I         *LOAD I
               C    1,=F'5'     *COMPARE WITH =F'5'
               BNL  Z           *BRANCH IF NOT LOW TO Z
               B    A           *BRANCH TO A
Z              EQU  *           *SYMBOLIC ADDRESS OF Z
               SVC  0           *RETURN TO SUPERVISOR
*
I              DS   F           *STORAGE FOR I
               END  P           *TERMINATE ASSEMBLY
```

Legal Program Q

```
Q:      PROCEDURE;
        DECLARE LX LABEL;
        L:      I = I + IA*IB - IC;
                LX = L;
                GO TO CHECK;
        M:      I = I + 1;
                LX = M;
        CHECK:  IF I< LIMIT
                THEN GO TO LX;
END Q:
```

Translation for Q

```
*
*  ASSEMBLER LANGUAGE PROGRAM FOR Q
*
               BALR 15,0        *SET REGISTER 15 AS BASE REGISTER
Q              USING *,15       *INFORM ASSEMBLER THAT R15 IS BASE REG
L              EQU  *           *SYMBOLIC ADDRESS FOR PL/I LABEL L
               L    1,I         *LOAD I
               L    3,IA        *LOAD IA
               M    3-1,IB      *MULTIPLY BY IB
               AR   1,3         *ADD REGISTERS
               S    1,IC        *SUBTRACT IC
               ST   1,I         *STORE RESULT IN I
               MVC  LX,=A(L)    *SET LX EQUAL TO ADDRESS OF L
               B    CHECK       *BRANCH TO CHECK
M              EQU  *           *SYMBOLIC ADDRESS FOR PL/I LABEL M
               L    1,I         *LOAD I
               A    1,=F'1'     *ADD =F'1'
               ST   1,I         *STORE RESULT IN I
               MVC  LX,=A(M)    *SET LX EQUAL TO ADDRESS OF M
CHECK          EQU  *           *SYMBOLIC ADDRESS FOR PL/I LABEL CHECK
               L    1,I         *LOAD I
               C    1,LIMIT     *COMPARE WITH LIMIT
               BNL  Z           *BRANCH IF NOT LOW TO Z
               L    1,LX        *LOAD ADDRESS STORED IN LX
               BR   1           *BRANCH TO THIS ADDRESS
Z              EQU  *           *SYMBOLIC ADDRESS OF Z
               SVC  0           *RETURN TO SUPERVISOR
*
LX             DS   F           *STORAGE FOR LX
I              DS   F           *STORAGE FOR I
IA             DS   F           *STORAGE FOR IA
IB             DS   F           *STORAGE FOR IB
IC             DS   F           *STORAGE FOR IC
LIMIT          DS   F           *STORAGE FOR LIMIT
               END  Q           *TERMINATE ASSEMBLY
```

Appendix 3 BACKUS-NAUR FORM SPECIFICATION OF SYNTAX OF LITTLE PL/I

1.	<letter>	::= A|B ... |Z	12.	<IF statement>	::= IF <boolean expression> THEN <non-DECLARE statement>
2.	<digit>	::= 0|1 ... |9	13.	<label assign statement>	::= <label variable> = <label>;| <label variable> = <label variable>;
3.	<identifier>	::= $[<\text{letter}>]_1^8$	14.	<arith op>	::= +| -|*| /
4.	<unsigned integer>	::= $[<\text{digit}>]_1^{10}$	15.	<arith exp>	::= <primary>|<arith exp><arith op> <arith exp>
5.	<fix-pt variable>	::= $[\text{I}\|\text{J}\|\text{K}\|\text{L}\|\text{M}\|\text{N}]_1^1[<\text{letter}>]_0^7$	16.	<arith assign statement>	::= <fix-pt variable> = <arith exp>;
6.	<label>	::= <identifier>	17.	<DECLARE statement>	::= DECLARE <label variable> LABEL;
7.	<label variable>	::= <identifier>	18.	<non-DECLARE statement>	::= <GO TO statement>|<IF statement>| <label assign statement>| <arith assign statement>
8.	<primary>	::= <unsigned integer>| <fix-pt variable>	19.	<statement>	::= $[<\text{label}>:]_0^1$ <non-DECLARE statement>| $[<\text{label}>:]_0^1$ <DECLARE statement>
9.	<GO TO statement>	::= GO TO <label>;| GO TO <label variable>;	20.	<statement sequence>	::= $[<\text{statement}>]_1^\infty$
10.	<rel op>	::= <|=|>	21.	<PL/I program>	::= <label>: PROCEDURE; <statement sequence> END <label>;
11.	<boolean expression>	::= <primary><rel op> <primary>			

Notation: In addition to pure Backus-Naur Form, we use the brackets $[\quad]_{k_1}^{k_2}$ to designate any number from k_1 through k_2 of occurrences of the enclosed expression.

Appendix 4 CANONIC SYSTEM SPECIFICATION OF SYNTAX OF LITTLE PL/I

1.	letter	$\vdash$ A ∤ B ∤ ... ∤ Z letter
2.	digit	$\vdash$ 0 ∤ 1 ∤ ... ∤ 9 digit
3.	identifier	$t_1 \nmid t_2 \nmid t_3 \nmid t_4 \nmid t_5 \nmid t_6 \nmid t_7 \nmid t_8$ letter $\vdash$ $t_1 \nmid t_1 t_2 \nmid \ldots \nmid t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8$ identifier
4.	unsigned integer	$d_1 \nmid d_2 \nmid d_3 \nmid d_4 \nmid d_5 \nmid d_6 \nmid d_7 \nmid d_8 \nmid d_9 \nmid d_{10}$ digit $\vdash$ $d_1 \nmid d_1 d_2 \nmid \ldots \nmid d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10}$ unsigned integer
5.1	first letter for fix-pt var	$\vdash$ I ∤ J ∤ K ∤ L ∤ M ∤ N first letter for fix-pt var
5.2	fix-pt var	t_1 first letter for fix-pt var $\nmid t_2 t_3 \nmid \ldots \nmid t_8$ letter $\vdash$ $t_1 \nmid t_1 t_2 \nmid \ldots \nmid t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8$ fix-pt var
6.	label	i identifier $\vdash$ i label
7.1	label var	i identifier $\vdash$ i label var
7.2	list	$\vdash$ Λ list
7.3		i identifier $\vdash$ i, list
7.4		a ∤ b list $\vdash$ ab list
7.5	in	a ∤ b ∤ c list $\vdash$ b in abc
7.6		a in $t \nmid$ b in $t \vdash$ ab in t
7.7	differ	$\vdash$ A ∤ A ∤ ... ∤ A differ B ∤ C ∤ ... ∤ Z $\vdash$ B ∤ B ∤ ... ∤ B differ A ∤ C ∤ ... ∤ Z . . . $\vdash$ Z ∤ Z ∤ ... ∤ Z differ A ∤ B ∤ ... ∤ Y
7.8		x ∤ y letter $\nmid$ x differ y $\nmid$ axd ∤ ayc identifier $\vdash$ axd differ ayc
7.9		x differ y $\vdash$ y differ x
7.10	disjoint	$\vdash$ Λ disjoint Λ
7.11		i identifier $\vdash$ i, disjoint Λ $\nmid$ Λ disjoint i,
7.12		i differ j $\vdash$ i, disjoint j,
7.13		x disjoint $t \nmid$ y disjoint $t \vdash$ xy disjoint $t \nmid t$ disjoint xy

Appendix 4 (continued) CANONIC SYSTEM SPECIFICATION OF LITTLE PL/I

8.1	primary with fix-pt var	i unsigned integer ⊢ i primary with fix-pt var Λ
8.2		v fix-pt var ⊢ v primary with fix-pt var v,
9.1	goto stm with ref label-label var	l label ⊢ GO TO l; goto stm with ref label l, label var Λ
9.2		l label var ⊢ GO TO l; goto stm with ref label Λ label var l,
10.	relop	⊢ < ∤ = ∤ > rel op
11.	boolean exp with fix-pt vars	p ∤ p' primary with fix-pt var v ∤ v' ∤ r rel op ⊢ prp' boolean exp with fix-pt vars vv'
12.	if stm with ref label-label vars-fix-pt vars	b boolean exp with fix-pt vars v ∤ s non-declare stm with ref labels l_r label vars l_v fix-pt vars v' ⊢ IF b THEN s; if stm with ref label l_r label vars l_v fix-pt vars vv'
13.1	label assign stm with ref label-label vars	v label var ∤ l label ⊢ v = l; label assign stm with ref label l, label vars v,
13.2		v ∤ v' label var ⊢ v = v'; label assign stm with ref label Λ label vars v, v',
14.	arith op	⊢ + ∤ - ∤ * ∤ / arith op
15.1	arith exp with fix-pt vars	p primary with fix-pt var v ⊢ p arith exp with fix-pt vars v
15.2		a ∤ a' arith exp with fix-pt vars v ∤ v' ∤ o arith op ⊢ aoa' arith exp with fix-pt vars vv'
16.	arith assign stm with fix-pt vars	v_1 fix-pt var ∤ a arith exp with fix-pt vars v ⊢ v_1 = a; arith assign stm with fix-pt vars vv_1,
17.	declare stm with decl label var	v label var ⊢ DECLARE v LABEL; declare stm with decl label var v,
18.1	non-declare stm with ref label-label vars-fix-pt vars	g goto stm with ref label l_r label var l_v ⊢ g non-declare stm with ref labels l_r label vars l_v fix-pt vars Λ
18.2		i if stm with ref label l_r label vars l_v fix-pt vars v ⊢ i non-declare stm with ref labels l_r label vars l_v fix-pt vars v
18.3		l label assign stm with ref label l_r label vars l_v ⊢ l non-declare stm with ref labels l_r label vars l_v fix-pt vars Λ
18.4		a arith assign stm with fix-pt vars v ⊢ a non-declare stm with ref labels Λ label vars Λ fix-pt vars v
19.1	stm seq with stm labels-ref labels-label vars-decl label vars-fix-pt vars	s non-declare stm with ref labels l_r label vars l_v fix-pt vars v ⊢ s stm seq with stm labels Λ ref labels l_r label vars l_v decl label vars Λ fix-pt vars v
19.2		s non-declare stm with ref label l_r label vars l_v fix-pt vars v ∤ l label ⊢ l: s stm seq with stm labels l, ref labels l_r label vars l_v decl label vars Λ fix-pt vars v
19.3		d declare stm with decl label var v ⊢ d stm seq with stm labels Λ ref labels Λ label vars Λ decl label vars v fix-pt vars Λ
19.4		d declare stm with decl label var v ∤ l label ⊢ l: d stm seq with stm labels Λ ref labels Λ label vars Λ decl label vars v fix-pt vars Λ
20.1		s ∤ s' stm seq with stm labels l_s ∤ l'_s ref labels l_r ∤ l'_r label vars l_v ∤ l'_v decl label vars l_{vd} ∤ l'_{vd} fix-pt vars v ∤ v' ∤ l_s disjoint l'_s ⊢ ss' stm seq with stm labels $l_s l'_s$ ref labels $l_r l'_r$ label vars $l_v l'_v$ decl label vars $l_{vd} l'_{vd}$ fix-pt vars vv'
20.2	legal stm seq with stm labels-vars	s stm seq with stm labels l_s ref labels l_r label vars l_v decl label vars l_{vd} fix-pt vars v ∤ v disjoint $l_s l_{vd}$ ∤ l_s disjoint l_{vd} ∤ l_r in l_s ∤ l_v in l_{vd} ⊢ s legal stm seq with stm labels l_s vars l_{vd} v
21.	PL/I program	l label ∤ s legal stm seq with stm labels l_s vars v ∤ l, disjoint l_s v ⊢ l: PROCEDURE; s END l; PL/I program

Appendix 5 CANONIC SYSTEM SPECIFICATION OF SYNTAX OF LITTLE PL/I

AND TRANSLATION INTO IBM SYSTEM/360 ASSEMBLER LANGUAGE

(Canons 1'. through 7.13'. same as canons 1. through 7.13. in Appendix 4.)

8.1'	primary with fix-pt var-operand code	i unsigned integer ⊢ i, primary with fix-pt var Λ operand code = F'i'
8.2'		v fix-pt var ⊢ v, primary with fix-pt var v, operand code v
9.1'	goto stm with ref label-label var-assembler stms	*l* label ⊢ GO TO *l*; goto stm with ref label *l*, label var Λ assembler stms B *l* *BRANCH TO *l*
9.2'		*l* label var ⊢ GO TO *l*; goto stm with ref label Λ label var *l* assembler stms L 1,*l* *LOAD ADDRESS STORED IN *l* BR 1 *BRANCH TO THIS ADDRESS
10.'	rel op with branch code-message	⊢ < ∤ = ∤ > rel op with branch code BNL ∤ BNE ∤ BNH message NOT LOW ∤ NOT EQUAL ∤ NOT HIGH
11.'	boolean exp with fix-pt vars-branch label assembler stms	p ∤ p' primary with fix-pt var v ∤ v' operand code x ∤ x' ⊣ r relop with branch code b message g ⊣ *l* label ⊢ prp' boolean exp with fix-pt vars vv' branch label *l*, assembler stms L 1,x *LOAD x C 1,x' *COMPARE WITH x' b *l* *BRANCH IF g TO *l*
12.'	if stm with ref label-label vars-fix-pt vars-assembler labels-assembler stms	b boolean exp with fix-pt vars v branch label *l* assembler stms s_a ⊣ s noa-declare stm with ref label l_r label vars l_v fix-pt vars v' assembler labels l_a assembler stms s'_a ⊣ *l*, disjoint l_a ⊢ IF b THEN s if stm with ref label l_r label vars l_v fix-pt vars vv' assembler labels l_a *l*, assembler stms s_a s'_a *l* EQU * *SYMBOLIC ADDRESS OF *l*
13.1'	label assign stm with ref label-label vars-assembler stm	v label var ⊣ *l* label ⊢ v = *l*; label assign stm with ref label *l*, label vars v, assembler stm MVC v,=A(*l*) *SET v EQUAL TO ADDRESS OF *l*
13.2'		v ∤ v' label var ⊢ v = v'; label assign stm with ref label Λ label vars v, v,' assembler stm MVC v,v' *SET v EQUAL TO v'
14.1'	add op with op code-message	⊢ + ∤ - add op with op code A ∤ S message ADD ∤ SUBTRACT
14.2'		* ∤ /multop with op code M ∤ D message MULTIPLY BY ∤ DIVIDE BY
15.1'	arith exp with fix-pt vars-result in reg-assembler stms	p primary with fix-pt var v operand code x ⊢ p arith exp with fix-pt vars v result in reg 1 assembler stms L 1,x *LOAD x
15.2'	term with fix-pt vars-result in reg-assembler stms	p ∤ p' primary with fix-pt var v ∤ v' operand code x ∤ x' ⊣ m mult op with op code c message g ⊢ pmp' term with fix-pt vars vv' result in reg 1 assembler stms L 1,x *LOAD x M 0,x *g x ⊣ pmp' term with fix-pt vars vv' result in reg 3 assembler stms L 3,x *LOAD x M 2,x *g x
15.3'		t term with fix-pt vars v result reg i assembler stms s_a ⊣ p primary with fix-pt var v_1 operand code x ⊣ m mult op with op code c message g ⊢ tmp term with fix-pt vars vv_1 result in reg i assembler stms s_a c i-1,x *g x
15.4'	arith exp with fix-pt vars-result in reg-assembler stms	t term with fix-pt vars v result in reg 1 assembler stms s_a ⊢ t arith exp with fix-pt vars v result in reg 1 assembler stms s_a
15.5'		a arith exp with fix-pt vars v result in reg 1 assembler stms s_a ⊣ p primary with var v_1 operand code x ⊣ o add op with op code c message g ⊢ aop arith exp with fix-pt vars vv_1 result in reg 1 assembler stms s_a c 1,x *g v_1
15.6'		a arith exp with fix-pt vars v result in reg 1 assembler stms s_a ⊣ t term with fix-pt vars v' result in reg 3 assembler stms s'_a ⊣ o add op with op code c message g ⊢ aot arith exp with fix-pt vars vv' result in reg 1 assembler stms s_a s'_a cR 1,3 *g REGISTERS

Appendix 5 (continued) CANONIC SYSTEM SPECIFICATION OF SYNTAX OF LITTLE PL/I AND TRANSLATION INTO IBM SYSTEM/360 ASSEMBLER LANGUAGE

16.'	arith assign stm with fix-pt vars-assembler stms	v_1 fix-pt var ϕ a arith exp with fix-pt vars v result in reg 1 assembler stms s_a ⊢ v_1 = a; arith assign stm with fix-pt vars vv_1, assembler stms s_a ST 1,v_1 *STORE RESULT IN v_1
17.'	declare stm with decl label var	v label var ⊢ DECLARE v LABEL; declare stm with decl label var v,
18.1'	non-declare stm with ref labels-label vars-fix-pt vars-assembler labels-assembler stms	g goto stm with ref label l_r label var l_v assembler stms s_a ⊢ g non-declare stm with ref labels l_r label vars l_v fix-pt vars Λ assembler labels Λ assembler stms s_a
18.2'		i if stm with ref label l_r label vars l_v fix-pt vars v assembler labels l_a assembler stms s_a ⊢ i non-declare stm with ref labels l_r label vars l_v fix-pt vars v assembler labels l_a assembler stms s_a
18.3'		l label assign stm with ref label l_r label vars l_v assembler stms s_a ⊢ l non-declare stm with ref labels l_r label vars l_v fix-pt vars Λ assembler labels Λ assembler stms s_a
18.4'		a arith assign stm with fix-pt vars v assembler stms s_a ⊢ a non-declare stm with ref labels Λ label vars Λ fix-pt vars v assembler labels Λ assembler stms s_a
19.1'	stm seq with stm labels-ref labels-label vars-decl label vars-fix-pt vars-assembler labels-assembler stms	s non-declare stm with ref labels l_r label vars l_v fix-pt vars v assembler labels l_a assembler stms s_a ⊢ s stm seq with stm labels Λ ref labels l_r label vars l_v decl label vars Λ fix-pt vars v assembler labels l_a assembler stms s_a
19.2'		s non-declare stm with ref labels l_r label vars l_v fix-pt vars v assembler labels l_a assembler stms s_a ϕ l label ⊢ l: s stm seq with stm labels l, ref labels l_r label vars l_v decl label vars Λ fix-pt vars v assembler labels l_a assembler stms l EQU * *SYMBOLIC ADDRESS FOR PL/I LABEL l s_a
19.3'		d declare stm with decl label var v ⊢ d stm seq with stm labels Λ ref labels Λ label vars Λ decl label vars v fix-pt vars Λ assembler labels Λ assembler stms Λ
19.4'		d declare stm with decl label var v ϕ l label ⊢ l: d stm seq with stm labels Λ ref labels Λ label vars Λ decl label vars v fix-pt vars Λ assembler labels Λ assembler stms Λ
20.1'		s ϕ s' stm seq with stm labels l_s ϕ l'_s ref labels l_r ϕ l'_r label vars l_v ϕ l'_v decl label vars l_{vd} ϕ l'_{vd} fix-pt vars v ϕ v' assembler labels l_a ϕ l'_a assembler stms s_a ϕ s'_a ϕ l_s disjoint l'_s ϕ l_a disjoint l'_a ⊢ ss' stm seq with stm labels $l_s l'_s$ ref labels $l_r l'_r$ label vars $l_v l'_v$ decl label vars $l_{vd} l'_{vd}$ fix-pt vars vv' assembler labels $l_a l'_a$ assembler stms $s_a s'_a$
20.2'	legal stm seq with stm labels-vars-assembler stms	s stm seq with stm labels l_s ref labels l_r label vars l_v decl label vars l_{vd} fix-pt vars v assembler labels l_a assembler stms s_a ϕ v disjoint $l_s l_{vd}$ ϕ l_s disjoint l_{vd} ϕ l_r in l_s ϕ l_v in l_{vd} ϕ l_a disjoint $l_s l_{vd}$ v ⊢ s legal stm seq with stm labels l_s vars l_{vd} v assembler stms s_a
21.1'	set of vars with assembler data stms	⊢ Λ set of vars with assembler data stms Λ
21.2'		v set of vars with assembler data stms s_d ϕ v_1 identifier ϕ v_1, disjoint v ⊢ vv$_1$, set of vars with assembler data stms s_d v_1 DS F *STORAGE FOR v_1
21.3'	PL/I program with translation	l label ϕ s legal stm seq with stm labels l_s vars v assembler stms s_a ϕ l, disjoint l_sv ϕ v' set of vars with assembler data stms s_d ϕ v in v' ⊢ l: PROCEDURE; s END l; PL/I program with translation * * ASSEMBLER LANGUAGE PROGRAM FOR l * BALR 15,0 *SET REGISTER 15 AS BASE REGISTER l USING *,15 *INFORM ASSEMBLER THAT R15 IS BASE REG s_a SVC 0 *RETURN TO SUPERVISOR * s_d END l *TERMINATE ASSEMBLY

Generalized translation of programming languages

by RONALD W. JONAS
Linguistics Research Center
The University of Texas
Austin, Texas

INTRODUCTION

A generalized model for the computer translation of both programming and natural languages has been developed at the Linguistics Research Center of The University of Texas. The design of the model is discussed here and sample grammatical descriptions are given to illustrate how the generalized translation of programming languages may be accomplished.

Discussion of the model

The linguistic theory of transformational grammar provides convenient terminology for explaining the LRC model. This theory maintains that there is not merely one structure underlying a language, but two: *surface* structure and *basic* structure. Viewed as a generative system, a transformational model is composed of a base component, which generates *basic* tree structures, and a transformational component, which maps these basic trees onto *surface* tree structures. The terminals of the resultant surface trees define strings of some particular language, such as FORTRAN or ALGOL.

If we think of the transformational model as primarily generative, the base may be regarded as the first phase of the process, generating structural trees explicitly stating linguistic relationships underlying language strings. These trees have terminal nodes, but they do not necessarily form a string which is meaningful in any particular language. The representation used by most linguists for the base is an ordered context sensitive grammar. Context sensitivity and ordering together control the type of structures output from the base. Figure 1 shows one such structure.

The transformational component, the last phase of the generative process, is expressed as a grammar having ordered rules in a highly specialized format. These rules specify how to take particular basic trees and reshape them so the terminals of the resultant trees are in the desired order. The resultant terminals must form a string, reading left to right, which belongs to the language for which the transformations are written (e.g. FORTRAN or ALGOL). For example, if the string *abcedf* in Figure 1 is not a string of language X, then a transformation may be specified to yield some surface tree with string *abcdef* belonging to the language X. One result of such a transformation is shown in Figure 2. In this case the transformation specifies that terminal branch *d* is to be moved from node *C* to node *B* of the tree.

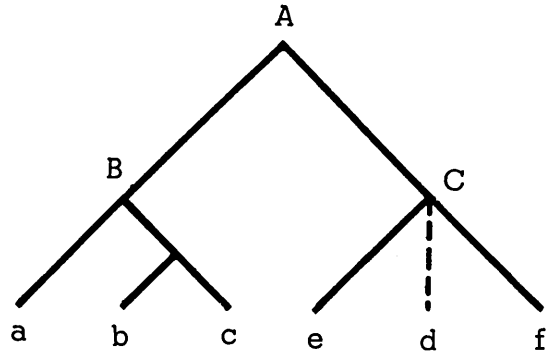


Figure 1—Basic tree

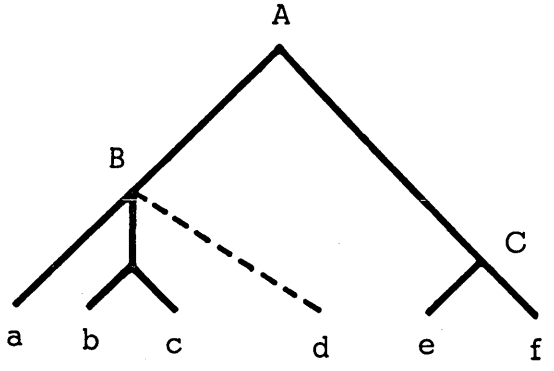


Figure 2—Surface tree

The transformational model details a mapping from base trees onto surface trees for only those parts of the two trees which are not identical, creating a rather economical statement of structure. A minimal statement is achieved for the generative process with base trees (formed by the base component) and a set of transformations (applied by the transformational component), which together yield a surface tree. Since transformations serve simply to reshape the basic tree into a surface tree, the mapping of base structure into identical surface structure is not explicitly stated.

Some linguists maintain that the base component characterizes language-independent organization while the transformational component characterizes language-specific organization. It is tempting to say that the base states semantic relationships and that the transformational component states syntactic relationships, but such a dichotomy is artificial at best. In such a model, syntax, as it is traditionally defined, refers to the structure explicitly stated by both the base grammar and the transformations. Semantics would then be defined as a combination of what is implicitly stated by the syntax and of the theory which led to the particular base and transformational rules in use. The semantics of language is explicit to the extent that the theory specifies algorithmically how to code the base and transformational component grammars. Perhaps it is better to say, for a particular language, that the surface structure *highlights* the syntax while the basic structure *highlights* the semantics.

By virtue of the power available in its pair of hierarchically-arranged grammars, transformational theory provides a system of importance to the mechanical translation of languages. To be sure, transformational theory itself does not solve the problem of translation, for it is concerned only with representing basic trees and mapping them onto the strings of some language(s). Translation must involve a reverse process as well, as suggested by Figure 3. For the compilation of programming languages, this diagram has a special interpretation. The input processes may be viewed as FORTRAN, ALGOL, etc. The output process may be using grammars of programming languages such as viewed as using grammars of particular machine languages such as CODAP, MAP, binary codes, etc.

Although the input and output processes would seem to be inverses of each other, there are some very definite reasons why they are not. The input process is attempting to guess what basic trees underlie the input strings. Box 1 guesses what surface trees are likely to be involved. Box 2 uses this best-guess information to control its guesses as to which basic trees might be involved. Given certain basic trees, the output process merely applies the transformations specified for the target language to yield surface trees (Box 4) and finally synthesizes the surface trees into output strings (Box 5).

There is, nevertheless, much similarity between the input and output processes. Boxes 1 and 5 are each applying surface grammars of their particular languages to produce surface trees and output strings, respectively. The two operations differ in that one is using its grammar to find all possible surface trees that may

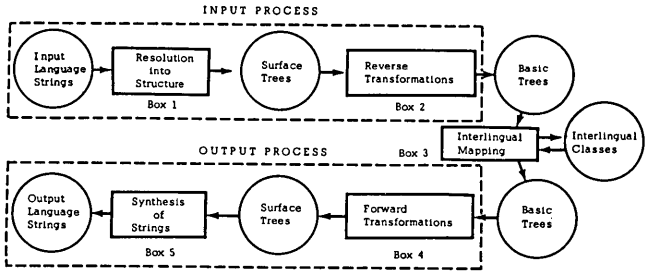


Figure 3—Generalized translation process

lead to the given input string while the other is given a particular surface tree and knows (from its grammar) exactly what string to produce from that tree. Analogously, Boxes 2 and 4 have their grammars of transformations. They differ in that one is trying to find all basic trees underlying the hypothesized surface trees while the other is given a particular basic tree from which to produce related surface trees.

Box 3 defines an interlingual mapping of input language basic trees onto output language basic trees. This process is best regarded as a two-stage operation: a subprocess which maps input basic structure into a set of interlingual classes, and a subprocess which maps these interlingual classes into output basic structure. These subprocesses are driven by grammars which associate basic substructures with certain interlingual classes. Those interlingual classes containing a single basic substructure for each language would provide *exact* translations, while classes containing multiple substructures per language would provide *loose* translations. For programming languages this would reflect the difference, for example, between FORTRAN statements which can be compiled into code only one way (e.g., CALL) and those which may be compiled any number of ways (e.g., A=−(B+C)).

A mechanical translation model much like the one discussed here is operational at the Linguistics Research Center. The specification languages of all boxes are roughly context free grammars, but ones which have operators available for each term of a rule. With these grammars it is possible to define and manipulate a terminal vocabulary in Boxes 1 and 5, to perform a

monolingual mapping between surface and basic trees in Boxes 2 and 4, and to scan trees and associate interlingual classes in Box 3.

Descriptive procedures

To accomplish generalized translation within the framework of this model, it is necessary to define a semantics of computing and to establish grammars suitable for intertranslating procedure-oriented languages, machine-symbolic languages, and machine codes. The present objective is to formulate descriptions which will be suitable for translating procedure-oriented languages and machine-smbolic languages into binary code. If the descriptions prove to be truly generalized, then any-direction translation will be possible; but the current objective is generalized **compilation.**

The usual procedure in generalized compiling is to apply a grammar to the input strings and then to engage semantic routines to interpret the results. The structural trees assigned to the input by the grammar contain nonterminal classifications which control the operation of semantic routines. Careful formulation of the grammars permits the proper semantic effects. The semantic routines build tables of semantic information which eventually yield the necessary machine code.

While certain current compilers have been designed to handle programming language syntax in a general way by accepting syntactic grammars, little has been done to generalize the information contained in the semantic routines of these compilers. The value of the model discussed here is that it permits complete grammatical description of both syntax and semantics. This allows an explicit statement of semantic classes in the form of a grammar rather than the implicit statement afforded by semantic routines. By providing the capability for coding a semantic grammar which captures the semantic information, this model eliminates the necessity for programming in a compiler and puts compilers, instead, completely in the realm of grammatical description of languages.

These are many problem areas requiring investigation before it is possible to write semantic grammars. The principal ones are as follows:

1. Data Representation
   a. Types of data; e.g., floating-point numbers, signed integers, alphabetic strings.
   b. Structural organization; e.g., matrices, lists, files.
   c. Forms of storage; e.g., bit strings, character strings, words, disk files, tape records.
2. Data Representation Operations. These would establish, change, and abolish data representations; exemplary operations: declarations, assignment statements for converting data types.
3. Data Manipulation Operations. This would include all operations upon any of the data defined in 2.
   a. General: add, test, truncate.
   b. Input-output: read, write, print.
   c. Transfers: move block storage.
4. Sequencing Operations. Any operations controlling the flow and synchronization of data manipulation operations would be in this group.
   a. Central processor: conditional and unconditional transfers, index jumps, central processor interrupts.
   b. Input-output: synchronization of independent data units, input-output interrupts.
   c. Console: communication interrupts.

All of these areas have been investigated preliminarily. The remainder of this paper will be devoted to a discussion of the data manipulation and sequencing operations.

As suggested above, discovering and grammatically describing basic trees is of great importance for this translation model. Of equal importance, however, is the necessity for defining the interlingual classes naming equivalent substructures of basic trees (Figure 3, Box 3). It does little good to define the semantics of each language to be translated unless a mapping can be guaranteed between the basic trees. Interlingual classes are used in defining this mapping. When a basic tree is partitioned into meaningful syntactic-semantic units, each block of the partition should be assigned to an interlingual class. If the corresponding basic trees entering and leaving Box 3 (Figure 3) are partitioned into equal numbers of blocks, then interlingual classes may be assigned to the blocks of the partitions in such a way as to define a 1:1 mapping between the input and output basic trees.

These interlingual classes are best defined before semantic grammars of particular languages are coded. Yet these classes can only be defined in a meaningful way by inspecting the types of basic trees they are intended to classify. Both will be discussed together in the following presentation.

Illustrative grammatical descriptions

The basic trees in Figures 4, 5, and 6 illustrate interlingual mappings for ALGOL, MAP, and some imaginary Machine X. Each figure contains basic trees which are equivalent and presumably translatable for the indicated languages. Solid lines define basic trees (such as the ones resulting from the application of grammars in both Boxes 1 and 2 of Figure 3). Broken-line enclosures are interlingual classes assigned to the substructure of the basic trees by the grammar of Box

3; the names of these interlingual classes are indicated as some T_x. The names themselves define the interlingual mapping of basic trees within each figure. See the Appendix for an explanation of MAP mnemonics.

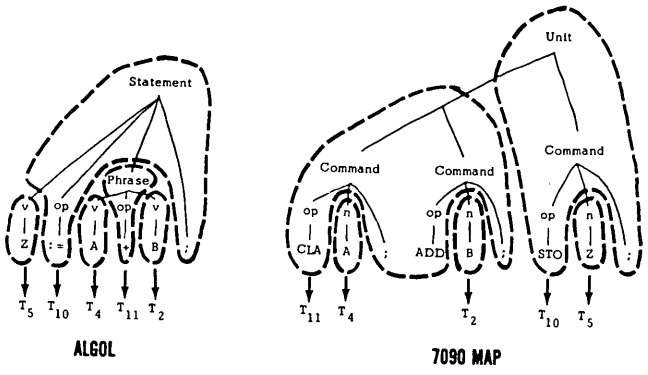


Figure 4—Structural mapping of addition operation

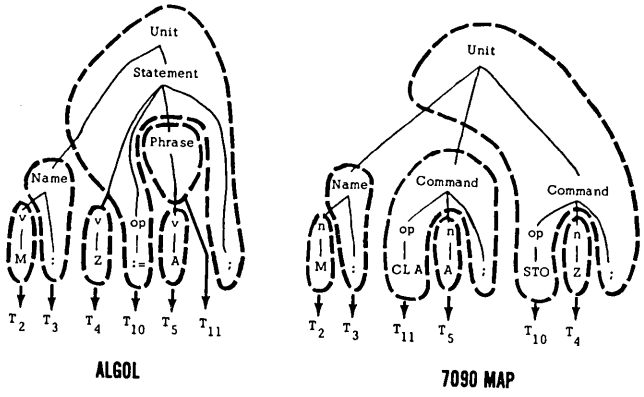


Figure 5—Structural mapping of assignment statement

As Figure 6 shows, the interlingual class T_1 serves to relate four equivalent operations within the three languages represented. Because the IBM 7090 is a single-address machine, more structure is included in the class T_1 for MAP than for either ALGOL or the multiple-address Machine Code X. In the same example, classes T_2,T_4, and T_5 provide an interlingual mapping of the variables involved in programming statements. As suggested in the earlier discussion of the translation model, these classes differ from other interlingual classes in that they may be defined during the translation process by special operators. For example, the intermediate symbols v in the ALGOL structures of the above examples may have associated operators which assign the lexical items *M, Z,* and *A* to unique interlingual classes (in this case, T_2, T_4, and T_5, respectively).

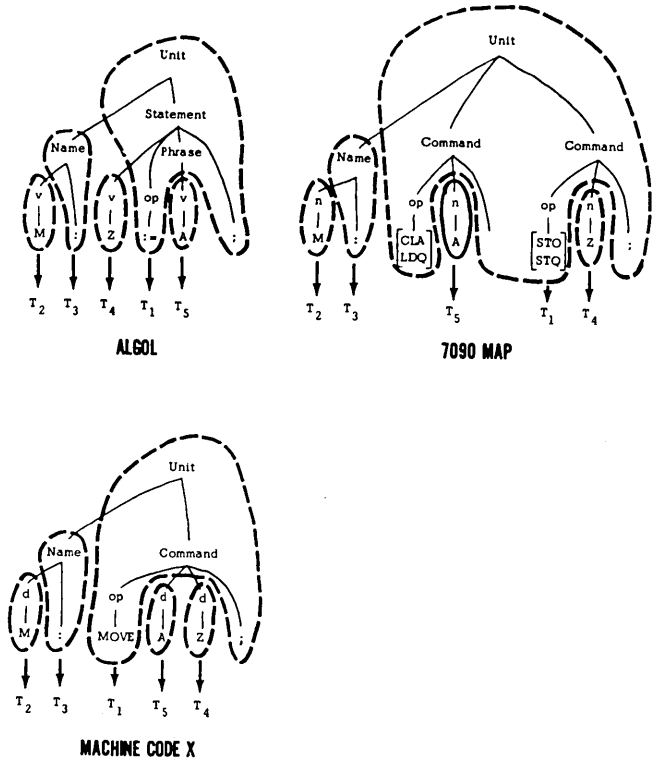


Figure 6—Structural mapping of data transfer operation. Terms within the two pairs of brackets are to be mutually selected. That is, CLA and STO co-occur or LDQ and STQ do. Colons and semicolons do not occur in MAP but are simply added for clarity (with ALGOL interpretations)

The relationships implied by the interlingual class names in these examples might be:

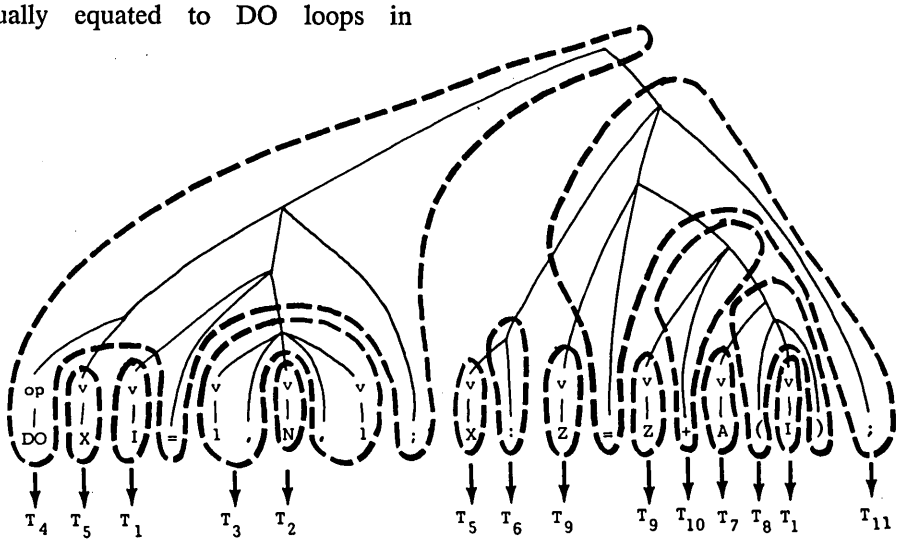
T_1 : Move data "statement"
T_3 : "Statement" name
T_{10} : Store value "statement"
T_{11} : Summation "statement"
$T_{2,4,5}$: map particular data structures from language to language

These classes explicate the semantics implied by the basic trees. For example, anyone who sees the symbol $:=$ in a statement quickly realizes it is an "assignment statement". He has mentally given a gross semantic classification to the part of the structure involving $:=$. One important fact that this overlooks is that the symbol ; was just as important in the definition of *assignment statement* as was $:=$.

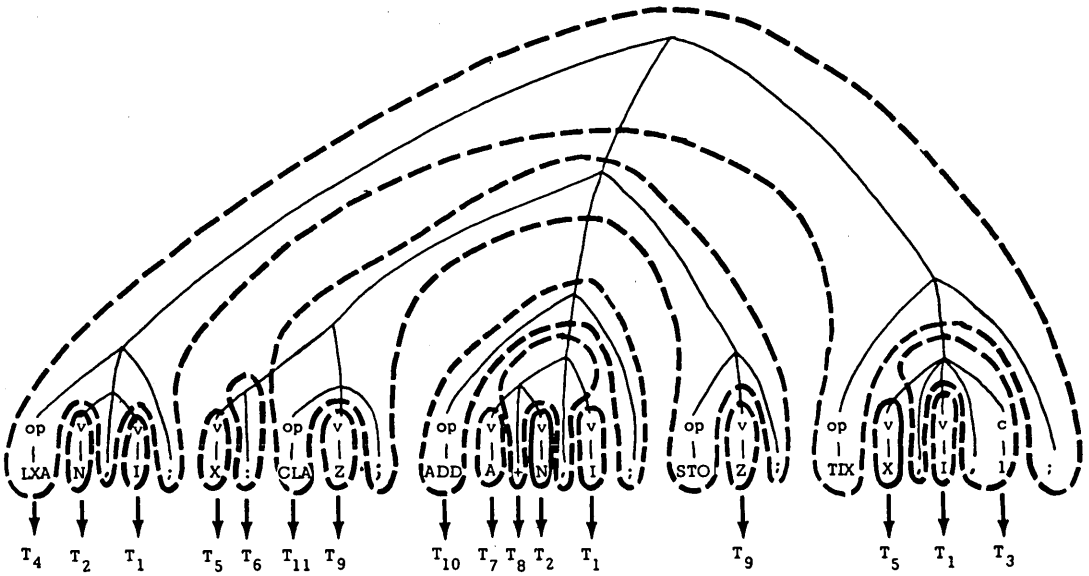
The classes named above are intended to reflect greater refinement of the semantic classifications. "Assignment statement" is too gross in the sense that it unites the semantic distinctions made by interlingual classes T_1 and T_{10}, i.e., the basic difference between moving a block of data into a new location (T_1) and storing the result of a summation (T_{10}). By more exact

isolation of structural information, the grammars which map basic trees into interlingual classes are able to specify explicitly all the semantic distinctions.

As might be guessed, there is no restriction on the size structure which might be included in an interlingual class. The smallest classes will consist of such things as variable names, some punctuation, and rather simple operations. The larger classes may be increasingly inclusive: programming loops in machine language may be interlingually equated to DO loops in FORTRAN or FOR loops in ALGOL; subroutines and even whole programs may become single interlingual classes or a set of such classes. For single interlingual classes to be semantically more inclusive, it is necessary to define larger basic trees. The only limit on such definition is the practical one of how inclusive the grammar is to be. For example, Figure 7 illustrates the basic trees for one possible DO loop to sum the elements of a matrix.



FORTRAN



7090 MAP

Figure 7—Structural mapping of basic trees underlying a DO loop

Clearly the MAP code used in Figure 7 is not the most efficient for representing the DO loop, although it is perfectly valid. This is due to the fact that the basic tree for the MAP statements is isomorphic with its corresponding surface tree. An improved sequence of MAP code is shown in Figure 8. The solid lines represent the surface structure imposed on the code. Dotted lines represent a transformation which maps that surface structure into the basic structure illustrated in Figure 9. Only the interlingual mappings which are different from those of Figure 7 are shown.

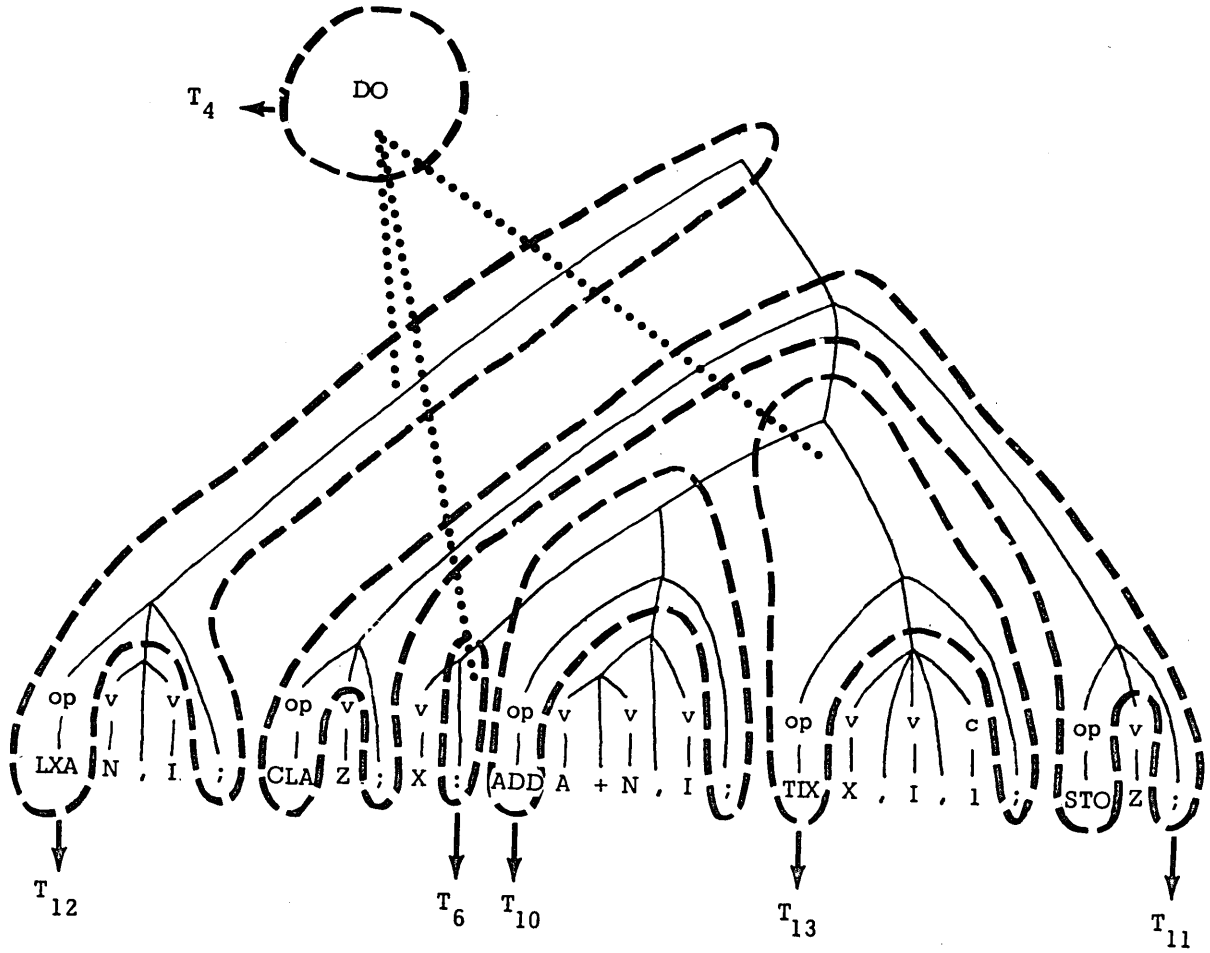


Figure 8—Structural mapping of surface tree for 7090 MAP code underlying DO loop

The surface tree (Figure 8) illustrates the improved MAP code in normal order. The basic tree (Figure 9) demonstrates the semantic functions isolated by the interlingual classes, as follows:

Section α : Opening and closing of DO loop
β : Name of loop
γ : DO loop proper

Clearly the basic tree explicates semantic information which is only implied in the surface tree. This is the motivation for investigating basic trees and interlingual classes rather than surface trees in defining the semantics of computing.

Data manipulation operations

With the advantages of basic trees in focus, it is timely to ask what the semantic classes should be and from what primitive units they should be built. The matter of semantic primitives is of most importance, for these will be the terminals of basic trees. For ALGOL the terminals might be operations such as :=+—/*. For MAP it might be ADD CLASTO, etc. If one were a computer engineer, the primitives of interest might be the operations of switching theory: *and, or, complement* and perhaps *nand, nor, exclusive-or.* Semantic primitives defined in this way appear not to

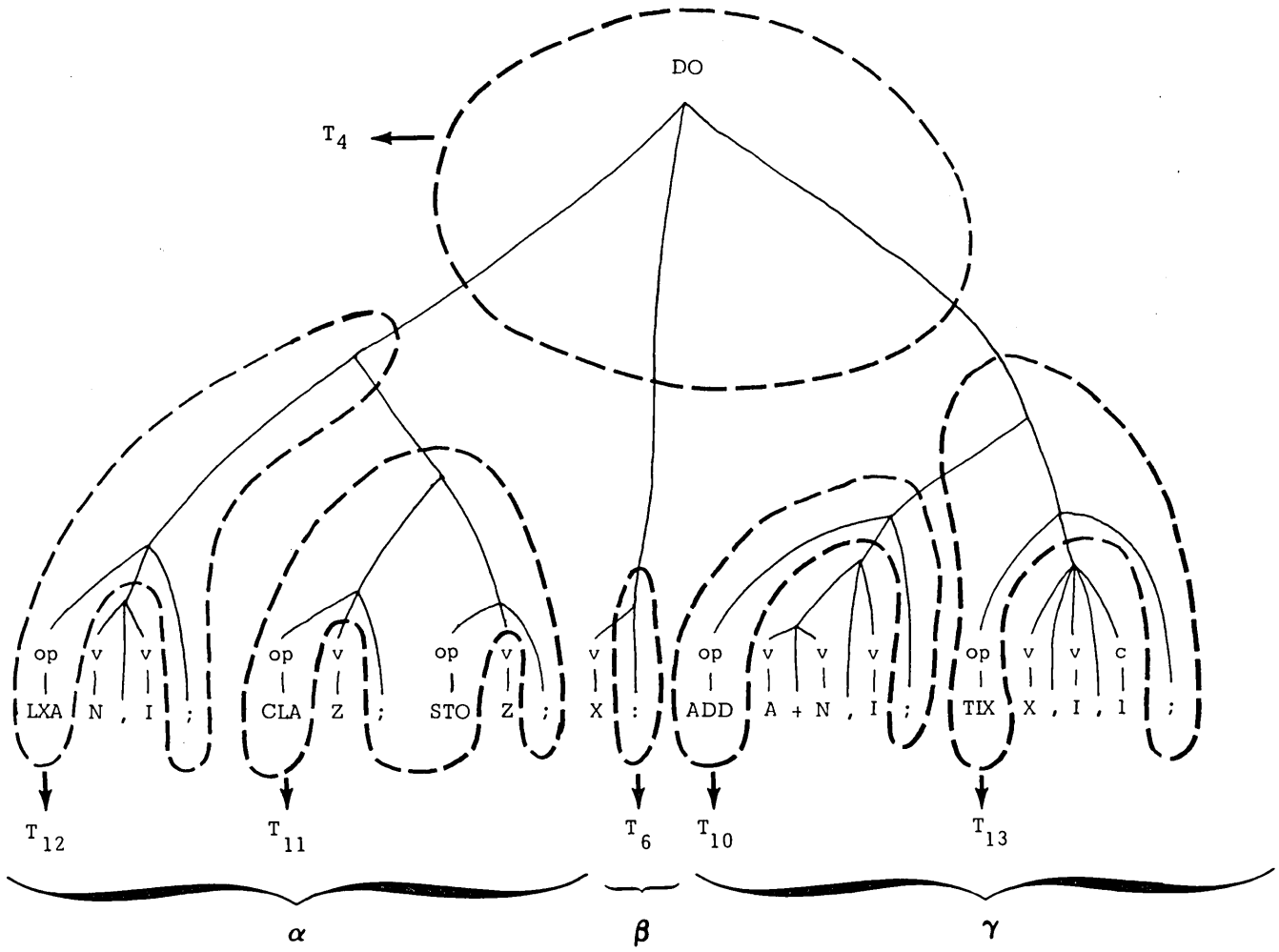


Figure 9—Structural mapping of basic tree for 7090 MAP code underlying DO loop

be generally applicable. One would not, for example, be happy with switching theory primitives as a basis for grammatical descriptions of ALGOL; and the ALGOL primitives would be entirely too gross for a grammatical description of sub-machine operations. In short, a separate set of semantic primitives must be defined for each language to be grammatically described.

Semantic primitives of this type, while useful for describing each language for which they are chosen, appear to have little interlingual value. The problem of mapping the primitives of one language onto those of another remains. This mapping is part of the more general problem of forming interlingual classes. Given that our initial task is the generalized compilation of programming languages, the mapping of primitives is an exercise in discovering the interlingual intersection of primitives for binary code and procedure-oriented or machine-symbolic languages. If we look at interlingual class T_{11} in Figure 4, we see that it contains the primitive operation $+$ for ALGOL. The corresponding MAP code for $\overline{T_{11}}$ is not at all primitive, however. Can the class T_{11} be called primitive? As suggested above, a given interlingual class is likely to be primitive only with respect to one of the two languages being inter-translated. This will be the language whose primitives are larger. Only when the two languages being translated are highly alike will there be any classes which are *primitive* with respect to both languages.

It appears, then, that the more useful process is that of finding, for two languages, which primitives must be defined in terms of sets of other primitives. For example, in Figure 4 the ALGOL structure mapped into interlingual Class T_{11} involves a single primitive. The MAP structure mapped into T_{11} involves the sequence of primitives *CLA ; ADD ;*. We can say

that the ALGOL primitive + is equivalent to the sequence of MAP primitives. This suggests an algorithm for creating interlingual classes; it would be applied to all languages in pairwise fashion: In general, the primitives of the more high-level language would define interlingual classes. The structure of the low-level language would be mapped into these classes as sequences of low-level primitives.

As an illustration of this process, assume we are translating MAP to 7090 binary code. The smallest interlingual classes would correspond to the primitive operation codes of MAP, such as *CLA, STO, ADD,* etc. While there is one complete 7090 binary code for each MAP operation, complete operations would not necessarily be the best primitives used in the grammar of the 7090. The 7090 operation code values reflect rather clearly the suboperations of the computer, and it is these suboperations which best serve as monolingual primitives of the binary code grammar. Table 1 is a partial list of MAP operations and the corresponding 7090 binary codes. The primitive suboperations indicated in Table 1 are explained in Table 2.

Description	*Mnemonic*	*Octual Code*	*Binary Code*
Multiply	MPY	0200	0 0 0 | 0 1 0 0 0 | 0 | 0 0 0
Variable-length multiply	VLM	0204	0 0 0 | 0 1 0 0 0 | 0 | 1 0 0
Divide	DVH	0220	0 0 0 | 0 1 0 0 1 | 0 | 0 0 0
Variable-length divide	VDH	0224	0 0 0 | 0 1 0 0 1 | 0 | 1 0 0
			D
Floating add	FAD	0300	0 0 0 | 0 1 1 0 0 | 0 | 0 0 0
Unnormalized floating add	UFA	4300	1 0 0 | 0 1 1 0 0 | 0 | 0 0 0
Floating add magnitude	FAM	0304	0 0 0 | 0 1 1 0 0 | 0 | 1 0 0
Floating subtract	FSB	0302	0 0 0 | 0 1 1 0 0 | 0 | 0 1 0
Unnormalized floating subtract	UFS	4302	1 0 0 | 0 1 1 0 0 | 0 | 0 1 0
Floating subtract magnitude	FSM	0306	0 0 0 | 0 1 1 0 0 | 0 | 1 1 0
Add	ADD	0400	0 0 0 | 1 0 0 0 0 | 0 | 0 0 0
Subtract	SUB	0402	0 0 0 | 1 0 0 0 0 | 0 | 0 1 0
			F *E* *B*
Place accumulator address in index	PAX	0734	0 0 0 | 1 1 1 0 1 | 1 | 1 | 0 0
Place accumulator decrement in index	PDX	4734	1 0 0 | 1 1 1 0 1 | 1 | 1 | 0 0
Place accumulator address in index comp.	PAC	0737	0 0 0 | 1 1 1 0 1 | 1 | 1 | 1 1
Place accumulator decrement in index comp.	PDC	4737	1 0 0 | 1 1 1 0 1 | 1 | 1 | 1 1
Place index in accumulator address	PXA	0754	0 0 0 | 1 1 1 1 0 | 1 | 1 | 0 0
Place index in accumulator decrement	PXD	4754	1 0 0 | 1 1 1 1 0 | 1 | 1 | 0 0
			C *A* *G*

Table 1. Partial list of operation codes for IBM 7090 computer. Certain suboperations are enclosed in boxes.

A Basic operation: multiply, floating point add-subtract, fixed point add-subtract, load into index, load from index

B Suboperation: add, subtract

C Word segment: address, decrement

D Length: word length, variable length

E Signing: signed value, magnitude

F Normalization

G Complementation

Table 2.'—Primitive suboperations chosen for IBM 7090 computer.

Clearly, the binary code is readily partitioned and yields a better set of primitives for the grammar of the machine than the full operation code. If each MAP operation defines an interlingual class, then the chart reveals the set of 7090 binary primitives which maps into each interlingual class. For example, the MAP operation *FAD* would be interlingually mapped together with the binary primitives *A. B. E, F* to cause translation into the complete binary operation 0300. An ALGOL operation would be mapped quite differently into the primitives, however. For example, the ALGOL addition operation makes no explicit statement about the variety of addition chosen. In the statement X : = Y + Z, the operation + would map into only the binary primitive *B* (Table 2). It would not involve the choice offered by primitives *A, E, F;* i.e., floating point vs. fixed, signed vs. unsigned, normalized vs. unnormalized. These parts of the binary operation could only be translated from information interlingually mapped with the variables X, Y, Z.

Sequencing operations

The grammatical description of sequencing operations is equally as important as the description of data manipulation operations. Such a description must account for the order in which data manipulation operations are executed and in which registers of the computer are addressed by these operations. The formulation of the description depends, once again, upon the definition of semantic primitives for sequencing.

In a computer like the IBM 650, every command explicitly states the location of the next command by an overt address. But in contemporary binary computers, sequencing is determined by a meta-process. That is, except for certain branching commands in the computer's repertoire, next addresses are determined by an addition process. There is an Address Register, much like the accumulator, which is stepped as each command is operated in order to cause sequencing to proceed linearly through memory. This process is depicted in Figure 10. It illustrates three alternate types of sequence control, which are characterized by the basic structures in Figure 11.

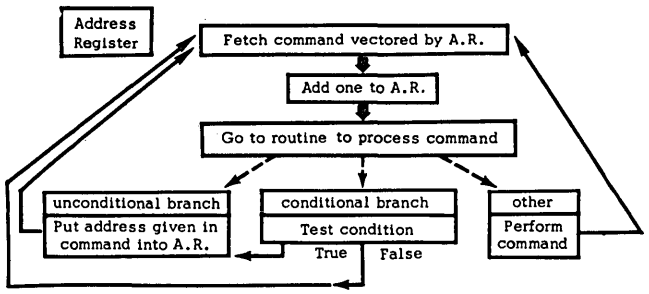


Figure 10—Conceptualization of meta-addressing in IBM 7090 computer

Ordinarily when we talk of computer operations, we refer only to what is going on in Boxes C (Figure 11). All of the hidden performance which goes into the processing of a sequenct of operation, however, is represented by the rest of the basic trees. These trees suggest a level of control in the computer which is superordinate to that of the operations proper, namely the control of sequencing.

While the above trees account for only the simplest variety of sequence control, related ones are capable of explaining a more difficult sequence phenomenon: interrupts. Each interrupt available on a computer is linked with some device which operates independently of the central processor. The key to the independent operation of each device and the behavior of its interrupt lies in Address Registers like the ones referenced in Figure 10 and 11. Each independent device may

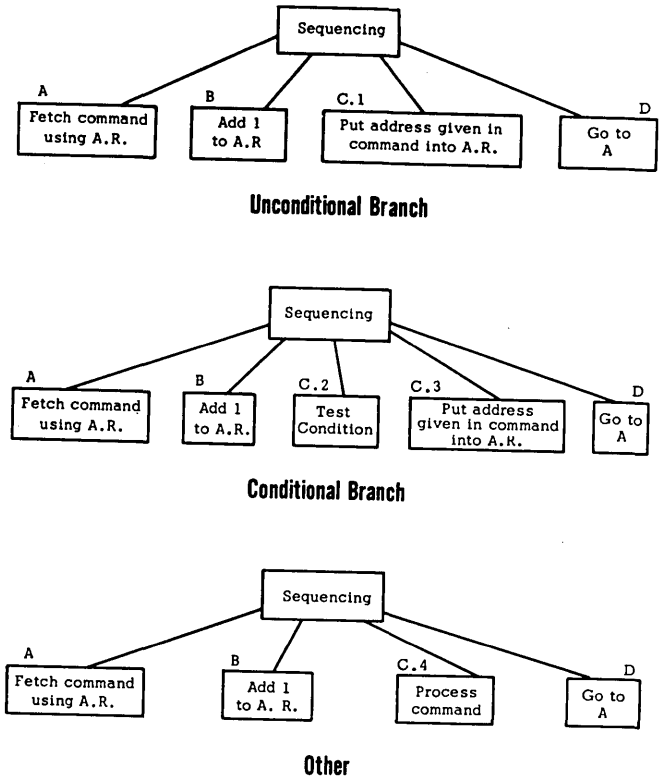


Figure 11—Basic structure for three types of sequence control in the central processor of the IBM 7090 computer

have an Address Register of its own to control its own sequencing. It may likewise have Address Register controls much like those in Figure 11, Boxes A and B. The device will only need such a control if it has a sequence of suboperations to perform. In any event, these devices are all coordinated with the central devise via its (central) Address Register.

As indicated in Figure 11, after each operation of the central processor is performed (Boxes C), control is returned to the central Address Register. Any time some independent device needs the services of the central processor, it may *interrupt* by altering the contents of the central Address Register while Box C is in progress for the central processor. Upon arriving at Box D (and subsequently A), the central processor will automatically be redirected by the contents of its altered Address Register. Therefore, we may account for even the interrupt behavior of computers by positing a basic structure for each interrupt. For example, the interrupt behavior of a typical independent device A might be characterized as in Figure 12.

A solution to the more general problem of addressing memory follows from this. The important consideration is that arithmetic may be performed on the

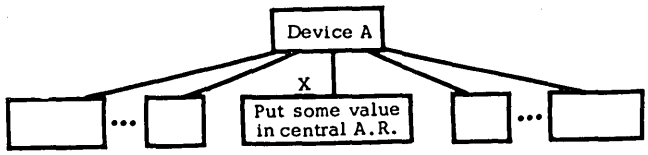


Figure 12—Basic structure underlying the interrupt behavior of Device A. Box X causes central processor control to be diverted

reference addresses themselves, not merely the data they reference. For example, every reference has potentially a base address and an index, which, when added together (or subtracted), yield the *effective* memory address. The execution of every command (Figure 1, Boxes C) having such an indexable address may be regarded as having a subcycle in which this effective address is computed, as shown in Figure 13. The 7090 has another such meta-operation. When more than one index is referenced by the same command, the logical sum of the indexes is taken and the result is then added (or subtracted) to the base address. This would require another subcycle on the far left of Figure 13 which accounted for the summing of indexes.

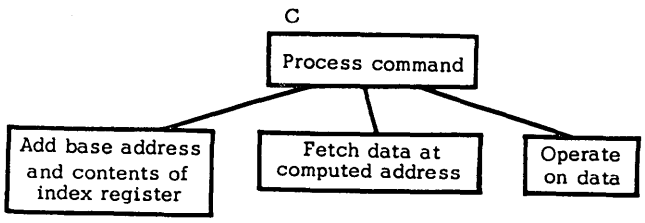


Figure 13—Basic structure underlying address indexing

In general, the grammars accepted by current compilers express relationships only within individual programming statements of the language being described. The relationships implied by the order in which these individual statements are executed is not explicitly coded. Compilers have not required or used this variety of information in the grammars. Data manipulation operations, on the other hand, involve well-defined functions for which convention has come to dictate fairly final grammars. Until grammars are written which state explicitly *all* the features of programming languages, *including* sequencing, translation of programming languages will not be general enough to be altogether semantically satisfying.

Optimization

The foregoing discussion has revealed that it is not enough merely to look at procedure-oriented languages as a means to formulating a theory of computing. The features of semantic grammars for procedure-oriented languages and for binary codes must be highly correlated if translation is to be realized. Such correlation is complicated by the fact that there is not always a single *best* way to map a procedure-oriented language such as ALGOL into the binary code for a particular computer. For example, one can define multiplication or division by powers of 2 in terms of either arithmetic or bit-shifting operations. A generalized compiler should be able to select either mapping. Furthermore, one would expect the compiler to select the code which leads to the shortest execution time for the given computer. If bit-shifting operations are faster but arithmetic operations are more general, then clearly the interlingual classes should include *both* to achieve the best translation of all multiplication and division statements into machine code. Inclusion of all such alternatives should be an objective for good semantic grammars.

SUMMARY

A model for language translation has been presented as a means for compiling computer languages. While in the same class as compiler building systems, it offers the advantage that the specification of programming languages and computer characteristics may be accomplished completely grammatically. In addition to providing generalized descriptive methods, the model offers the potential of free inter-translation of languages once grammars are coded.

An approach to the writing of grammars for programming languages is outlined. The suggestion is made that currently available syntactic grammars may be considered adequate and that effort should be put into the coding of semantic grammars. Major problem areas requiring investigation to this end are: ways of representing and storing data, operations for data representation and manipulation, and sequencing control. Exemplary grammatical descriptions are shown.

Particular attention is given to the description of data manipulation and sequencing operations. The discussion includes consideration of both the monolingual and interlingual problems of grammars for procedure-oriented and machine-symbolic languages and binary machine codes. The definition of a semantics of computing involves, consequently, not only determining what machine code is substituted for programming statements but also what the meanings of statements and sequences of statements are in both languages and what their common structural intersection is (interlingual classes). This has led to some tentative conclusions about what units are needed in a semantic theory in order to completely describe programming languages.

APPENDIX: IBM 7090 MAP Operations

Mnemonic	*Explanation*
ADD X	Add contents of X to accumulator
CLA X	Clear accumulator and add contents of X
LDQ X	Load Q register with contents of X
LXA X, Y	Load index register Y with address of word at X
STO X	Store accumulator contents at X
STQ X	Store Q register contents at X
TIX X, Y, Z	Reduce index register Y by Z amount; if result is 0, transfer to location X

BIBLIOGRAPHY

1 E W BACH
An introduction to transformational grammars
Holt Rinehart and Winston Inc New York 1964

2 N CHOMSKY
Aspects of the theory of syntax
The MIT Press Cambridge 1965

3 R S GAINES
On the translation of machine language programs
Communications of the ACM 8:736-741 December 1965

4 *IBM 7090 Principles of operation*
IBM Systems Reference Library File No 7090-01 Form A22-6528-5 August 1963

5 R K LINDSAY
Inferential memory as the basis of machines which understand natural language
Computers and Thought Edited by E. A. Feigenbaum and J Feldman McGraw-Hill Book Co New York 1963 pp 217-233

6 J D McCAWLEY
Concerning the base component of a transformational grammar
Ditto Revised University of Chicago August 16 1966

7 M R QUILLIAN
Semantic memory
Bolt Beranek and Newman Inc Cambridge Scientific Report no 2 October 1966

8 W A SASSAMAN
A computer program to translate machine language into FORTRAN
AFIPS Conference Proceedings 28:235-239 Spring Joint Computer Conference 1966

9 *Thirteenth quarterly progress report*
Linguistics Research Center The University of Texas at Austin pp 22-34 1 May 1962-31 July 1962

10 A M ZWICKY et al
The MITRE syntactic analysis procedure for transformational grammars
AFIPS Conference Proceedings 27:317-326 Fall Joint Computer Conference 1965

Computer change at the Westinghouse Defense and Space Center

by W. BARKLEY FRITZ*
Westinghouse Electric Corporation
Pittsburgh, Pennsylvania

INTRODUCTION

The Westinghouse-Baltimore Defense and Space Center is a military contractor of approximately 13,000 employees with sales billed in the neighborhood of one-quarter billion dollars annually. Computer operations are centralized with research, engineering, management science, and business data processing all being handled by the same facilities located in an Administrative Services Building nearby the two major Divisions of the complex but up to 60 miles away from other operations which it serves. At the present time the central computer is a UNIVAC 1108 with four connected UNIVAC 1004's, eight IBM 1050's and additional Friden Collectadata equipment which permit access to the 1108 from various remote and not so remote locations.

A status report on these facilities, however, is not the subject of this paper. My subject is *computer change* and in particular computer change from the point of view of an organization which has undergone fairly frequent and successful change.

The basic reasons for computer change are to *reduce costs* and *provide adequate computer capacity*. The demands of the rapidly changing technology of the defense business and the needs of its customers for machine processed information sometimes seem almost impossible to satisfy. However, it does not follow that the cost of computer processing must continue to climb. In fact, at the Westinghouse Defense and Space Center total costs for computer service have actually been reduced by nearly 40 percent during the past two years. This reduction has taken place in spite of the fact that the requirement has continued to grow. As a rule-of-thumb, by taking advantage of improved equipment, exploiting improved software and techniques and by more effective loading of equipment, it is possible to reduce per job computer costs by a factor of two every two years. In fact, the Westinghouse Defense and Space Center has used this rule-of-thumb as a guide for evaluating its performance over the past six years.

Computer acquisition at Westinghouse

At the Westinghouse Electric Corporation, we have developed over the years a fairly advanced computer acquisition procedure of which we are justly proud. We have centralized computer ordering. We treat the acquisition of computers as cost reduction projects. Divisions of the Corporation desiring computers are required to conduct a feasibility study culminating in a justification report. A procedure exists requiring the approval of the Division Manager, the Headquarters Management Systems Department, the Group Vice President to whom the Division reports, and for the most expensive models, the Corporate Capital Expenditures Committee. Our Headquarters Management Systems Department contains experienced computer specialists who assist the divisions in such deliberations. Reports have been issued describing our computer acquisition procedure, the cost reduction guidelines which must be met, and a checklist of items to be considered in the feasibility study and documented in the justification report. Among the items covered are the requirements that more than one computer supplier be considered and that the purchase or rent question be evaluated.

Additionally the Headquarters Management Systems Department negotiates contracts with the various computer manufacturers. These negotiations have resulted in several contractual conditions which have made computer change easier for the Westinghouse Divisions. These contractual conditions include:

1. 90-day cancellation option for complete systems;
2. 30-day cancellation option for individual units or subsystems;
3. Letters of intent and "pool" orders to facilitate delivery schedules; and

*The author was until recently the Manager of Information Systems and Programming at the Westinghouse-Baltimore Defense and Space Center where he had responsibility for computer planning and programming.

4. Software modifications or special software requirements to meet local needs.

Through the use of cost reduction guidelines for computer projects and additional contractual conditions, Westinghouse management has allowed healthy growth of its computer systems and their usage, while continuing to maintain effective control of the equipment.

Management guidelines

Efficient and effective change requires adherence to certain *management guidelines* on the use of the computer systems. Some of the more important of these guidelines in effect at the Baltimore Defense and Space Center are as follows:

1. *Machine independent programming languages are used* with careful attention to modularity in system design. *Subroutines* are constructed in such a way that they can be used in more than one application without change.
2. *Data files* are established in a standard machine processable format so that these files can be effectively maintained and used for more than one information requirement.
3. *Follow-up analyses are performed as a standard part of any equipment acquisition as well as for each application implemented.* All too frequently a computer based information system is designed to provide information to solve a particular problem. It is essential that the information should stop flowing when the original problem has been solved. Exception reporting should be the rule rather than the exception. The emphasis and format of computer output should continue to change as necessary to respond to changing management needs.
4. *Backup arrangements are provided for all equipment.* It is generally not economically feasible to have sufficient in-house equipment to meet all emergency situations, but one can usually have a neighbor with compatible equipment that can serve as mutual backup. Organizations such as users associations help to foster individual local arrangements. Sharing of ideas as well as equipment can also result from such backup arrangements. Eventually with machine independent programming, it won't be necessary for the backup installation to have the same type of equipment.
5. New types of ADP Equipment should never be acquired until such time as an extensive *workload has been established on Service Bureau* or other equipment elsewhere. This means that the computer programs required will have been written and tested before new equipment is installed.
6. Follow-up of equipment and individual applications is emphasized via the project schedule technique. Evaluation and *monitoring of the total load* is also performed. Charts indicating hours of usage, number of runs being made, hours per shift of "distributable" use, and related usage figures are maintained and carefully examined to ascertain trends. The amount of time for re-runs and set-up is monitored and action taken to control any undesirable trend. Unlike other service costs, computer service costs per unit of work, when properly controlled, do go down.
7. USASI sponsored standardization efforts are carefully followed to be certain that the internal standards policy is consistent with USA Standards. Proposed USASI standards are regularly published in the Communications of the Association for Computing Machinery.
8. *New problem solving techniques are communicated to individuals* responsible for implementing new computer programs. We must be certain that all of our system design work and each new computer program is properly reviewed by senior personnel to check accuracy, conformity to standards and to make certain that poor techniques are not being used. Programmers don't like this, but other work is reviewed, why not computer programs? Individuals are given responsibility for the quantity and quality of their work. They are informed as to the total requirement and the individual Analyst/Programmer is made *responsible* for his individual effort.

Westinghouse-Baltimore experience

To be more specific now with respect to the Westinghouse-Baltimore Defense and Space Center, I have included three figures which illustrate our growth in computer usage and improvement in price performance. Figure 1 shows the decline in typical job cost. Figure 2 illustrates the reduction in average cost per equivalent IBM 7094 hour over the past six years. In spite of, or rather in part because of, a growth from 10,000 computer passes per month in 1962 to over 25,000 passes at the present time as illustrated in Figure 3, the average cost per typical job and the average cost per equivalent computer hour have easily met the goal of a factor of two cost decrease every two years.

During the past four years over 40 changes have been implemented to the Westinghouse-Baltimore facilities. Some of the changes involved only individual units or special features; however, during this period 12 computer systems were released and replaced by 10 new systems. One of these changes was for a mechanical replacement of a 1401 system which was replaced by

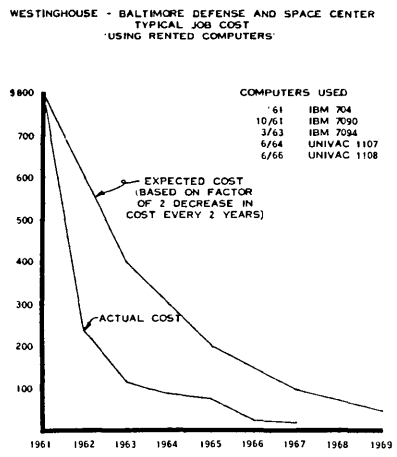


Figure 1—Westinghouse-Baltimore Defense & Space Center typical job cost (Using Rented Computers)

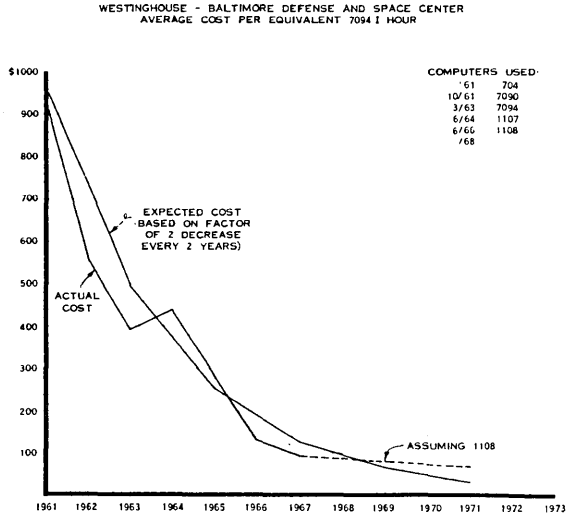


Figure 2—Westinghouse-Baltimore Defense & Space Center average cost per equivalent 7094 I hour

IBM when it could no longer be maintained. Most of the other changes were upgrading of 1401's to 1460's and finally to 360/30 systems. During the 4-year period, six 1401 systems and three 1460 systems were replaced or released. During the same period an IBM 7090 was upgraded to a 7094 which was later released. A UNIVAC 1107 was acquired and later replaced by an 1108. The present configuration consists of an 1108 and three IBM 360/30 systems.

Everyone of these changes was fully justified and controlled, as discussed previously, as a cost or expense reduction project. As I have attempted to show, computer change has made possible significant per job reductions in computation costs and provided desirable adjustments in the configuration of equipment as necessary to meet changing needs.

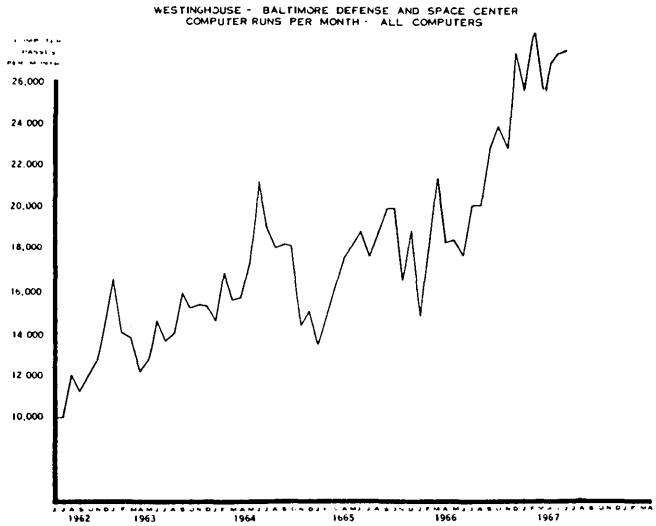


Figure 3—Westinghouse-Baltimore Defense & Space Center computer runs per month—all computers

I hope I have set the stage properly. *Computer change* shouid occur because newer equipment makes it possible to do the same load or an increased load at lower cost. I have tried to demonstrate that the Westinghouse-Baltimore Defense and Space Center has been able to make frequent change and has reaped the financial payoff from such change.

Attitude for change

Change must be supported by the right *attitude*. Essentially the attitude which supports change seems to be common among individuals who have been connected with computers for at least five years, sometimes much longer. Many of us were using such equipment even before the prevalence of magnetic core storage. We have lived through extensive technological change. At some time in the past we may have felt "stuck" with equipment beyond its technological or economic life because of the reprogramming problem. However, we are now going to make certain that we will not be "stuck" again. During the past several years we have insisted on the use of machine independent programming languages. We have recognized that change. At some time in the past we may have felt changes in existing operating programs to meet changing requirements. In being successful, our work load has grown through the introduction of new applications. We have a record of frequent change of our rented equipment because we have long ago recognized that modular hardware existed and that equipment could grow and change to meet new requirements. Although users, we have helped to force upon the computer manufacturer some of the hardware and software im-

provements currently available. Our ideas and needs have contributed to the new generation of equipment now being made available. In summary, we "liberals" of the computing field are prepared to make rapid use of the new equipment. Change for us means an ability to get more throughput per dollar, to move into areas heretofore untouched and to help make ADP an increasingly important part of the entire organizational picture.

State-of-the-art of program conversion

As further background, or stage setting, for my simple "formula for change," I would like to review briefly the state-of-the-art in the area of computer system change and reprogramming.

In June, 1965, the ACM sponsored jointly with Applied Data Research, Incorporated a "Reprogramming Conference." The three-day meeting was held in Princeton, N. J., and ten of the papers presented were published in the December, 1965 issue of the Communications of the ACM. For those interested in the reprogramming problem, this issue of CACM and the many references presented is a useful starting point.

Following is a brief summary of what was said at that conference—in essence an abstract of the abstracts:

1. A large file maintenance and retrieval system was written in COBOL and run successfully on three different IBM computers (1410, 7080, and 7090).
2. Completely automatic translation of machine language programs, although highly desirable, has not been achieved. Problems are primarily of a semantic nature which can be resolved by a semi-automatic procedure.
3. A set of macro-operations was used to assist in translating from the 7090 to the 7040 by inserting this set of macros at the beginning of the 7090 symbolic deck.
4. An RCA 301 "emulation" system was used to enable 301 object programs to be run on the Spectra 70/45. The emulation concept makes use of both hardware micro-program routines and software to accomplish the required action.
5. The emulation concept was further described as it applies to running 7074, 7080 and 7090 programs on the IBM System/360.
6. A translation system was described to eliminate "most" of the effort formerly required to reprogram Philco 2000 programs for operation on the IBM 7094.
7. Techniques utilizing a meta-language to map from one assembly language into another were described. Timing problems and "perverse" use of instructions presented difficulties.
8. The 1401 compatability feature for the IBM System 360 Model 30 was discussed. This paper covered what proved to be the most widely used computer conversion aid during the past two years. As most of us know, the read only storage used on the 360/30 has made the running of 1401 programs on the 360/30 a highly successful and economical venture.
9. A variety of special translation programs have been written to aid in the translation of programs from one version of FORTRAN to another and from one computer to another. LIFT and SIFT are two such programs. CAT (Computer Aided Translation) was a CDC 3600 programming system designed to aid in the translation of IBM 7090 FAP programs into 3600 COMPASS language programs.
10. A final survey paper considered hand recoding, automatic machine language to machine language translation, decompilation, meta-assembly (a generalized assembly program that accepts as input both symbolic instructions to be assembled, and parameters that in effect specify the machine for which they are to be assembled), and computer transference.

Not included in the wrap-up paper were the techniques used to translate very similar languages (e.g., FORTRAN II to FORTRAN IV), the quasi hardware "emulator" concept for translating existing application programs (both discussed in other papers), or the very generalized UNCOL concept which requires more unanimity among computer manufacturers and users than seems achievable.

A number of computer conversions have been recently reported, e.g., Doug Williams' article in the January issue of Datamation entitled "Conversion at Lockheed Missiles and Space." Another aid in converting was reported in February Datamation by Don Herman, of Comress, Incorporated, which is now marketing TRANSIM for "translating programs" from one machine to another. This is apparently a 100% effective translation for certain selected pairs of machines.

Management responsibility for change

The successful Westinghouse-Baltimore approach (our formula for change) is based on a fundamental tenet of management policy, i.e., the computer facility will replace existing hardware with new improved better price performance hardware as it becomes available. Changes, as outlined previously, did take place so

computer users have come to expect change and be ready for it.

In Westinghouse-Baltimore, personnel involved in the development of information systems have been also "exposed" for some time to a Department Manual which includes an introduction that emphasizes flexibility, and I quote, "a policy of obtaining the most effective processing equipment (from a cost per unit computation point of view) and using this equipment to its maximum potential." Such a policy requires the use of programming practices which minimize the cost of conversion to new equipment, thus the necessity for so-called machine independent programming.

The Department "goal" and "responsibilities" again emphasize these points and the following statement of programming policy puts added teeth into the management emphasis.

"FORTRAN/BEEF programming policy

"The basic programming policy is that all programming be accomplished as described in "The Compleat Guide to FORTRAN/BEEF Programming," published by the Westinghouse Defense and Space Center. This document provides the specification of this machine independent programming approach and documents acceptable programming practices and techniques.

"The following "Statement of Programming Standards" will be rigidly adhered to:

1. FORTRAN will be used as *the* programming language for all scientific, engineering, management science and data processing applications.
2. When conventional FORTRAN statements are basically inadequate, already prepared standard subroutines (BEEF) will be used via the CALL statement.
3. If the existing subroutine library is not adequate to accomplish the required result, a new subroutine will be defined, specified, programmed, and added to the BEEF library. This new routine will be written in FORTRAN and/or the symbolic language of the computer being used. In order to maintain machine independence the new subroutine will be written for any other computer on which the program is to be run, in other words the BEEF library will be maintained.
4. As an aid in debugging new programs, the BEEF editor will be used.
5. All programs will be written so that all referenced files may be changed among tapes, card readers, drums, discs, card punches, and printers by an appropriate change of unit number.
6. Careful attention must be given to make certain that any change to a new computer, a new executive system or exploitation of major increases or decreases in hardware configuration can be accomplished without difficulty.
7. Adequate comments will be included to make certain that the intent of the program is understood by others trained in FORTRAN programming.

"You Can Be Sure If It's Westinghouse" that the present computer will be replaced by a new system in the near future."

Obviously, a management policy does not mean computer conversion is automatic. A single programming language is not the complete answer, for although the basic programming language may be the same on two computers, there are frequently operating system differences. This facet of the problem can be solved by computer editing of input decks prior to their compilation. For example, it was possible for Westinghouse-Baltimore to run IBM 7094 FORTRAN program decks, using the IBSYS control cards, directly on the UNIVAC 1107. This was accomplished by having the control cards automatically converted by the 1107 on input. A few ground rules established in 1963 provided the basis for a standard FORTRAN. The burden was placed on the new system to behave like the old. This approach to handling the minor differences in operating systems is probably in fairly common use.

A problem in computer system conversion is that the user is only being partially supported by the supplier. The supplier is willing to provide assistance in converting from a competitor's equipment to his own, but is really not interested in helping the user to become independent of all suppliers, including himself. It would appear that today's operating systems are designed to help lock you into the brand you are now using. Again, the solution is in the hands of user management who must take steps not only to establish policy as noted earlier but in following through to see that machine independence is maintained. In-house operating systems capability is a must for effective follow-through.

Another facet of the "reprogramming" problem is the fact that many of the required major information systems in wide use were written for a particular computer by the supplier, e.g., PERT/COST, or a specific Design Automation System. The supplier of the equipment probably has written the major application system in symbolic machine language and unless forced to do so the supplier of the new equipment will not readily agree to rewrite the system. Contract negotiation time is the point when such problems must be resolved. The production of generalized software to meet user specification is one area where the computer supplier will

assist, particularly if that software is of general use. Westinghouse-Baltimore has had no trouble, for example, in getting the BEEF subroutines rewritten for new equipment. McDonnell and IBM have collaborated in the rewriting of BEEF for the 360 systems. CDC has included most of the BEEF subroutines in their software support and, of course, UNIVAC maintains this package for the 1107/1108.

Another problem is getting to and maintaining computer independence in programming is the attitude of the programmer himself. Unless very carefully trained and "managed" he will not automatically adhere to computer independence concepts. Consciously or unconsciously, he doesn't want to write programs that others can understand and change or adapt to new requirements. He seems to fear being not needed on his "creation." He tries to place his own personal touch on his programs. At Westinghouse we instill in our programmers the concept that the opportunity for advancement is in a large way dependent on their ability to keep their work machine independent by careful adherence to standardized routines and by careful adequate documentation. A complete description of standard documentation procedures has been included in the Department Manual referenced earlier. If the programmer doesn't do the job in this respect, he is released or becomes relegated to the role of a clerk or maintenance programmer with little opportunity for promotion or for involvement in the development of new systems.

Computer system change should also be examined from the point of view of the level of development of the applications being run on the existing system. Briefly as we see it in Westinghouse, there are four levels of application development:

1. *Observation*—Standard data reduction or data processing, the traditional data in, processed data out approach.
2. *Predictive simulation*—First level in which the information system or the mathematical model is used for forecasting purposes; sometimes called the "what if" game.
3. *Inversion*—The problem of predictive simulation is turned around in order to achieve a "design" having a specified performance.
4. *Optimization*—At the highest level, the concern is to obtain a "best" solution from among "all" designs providing a satisfactory performance.

Computer system change in a large way depends on the application level at which an organization is currently operating; complete resystematizing and reprogramming may actually be desirable in order to provide the impetus or opportunity to move up to a higher level at the same time as hardware change.

Personnel attitude toward change and the requirements and competitiveness of the industry in which the organization functions are other major factors affecting such action.

CONCLUSION

All of these problems, however, are overshadowed by the reductions in cost per job that computer change allows. With the right attitude, the proper management support, and proper planning, computer change can be highly successful.

I have attempted to point out that computer change is a full time continuing problem. Change to the next system must be considered even while implementing today's system change. Management policy must support *change* and all levels must work together to see that change can occur economically, quickly and effectively to satisfy the changing requirements of today's dynamic economy.

Machine-independence and third-generation computers

by MAURICE H. HALSTEAD
Lockheed Missiles & Space Company
Sunnyvale, California

and

Purdue University
Lafayette, Indiana

INTRODUCTION

As a man with keen insight noted when third-generation equipment was first announced, the problems posed by this equipment have only one reasonable solution: "Wait till you have the new equipment before setting up your computer center." Accepting that point of view, it follows that those of us working with existing computation facilities must concern ourselves with somewhat less reasonable solutions and more difficult approaches. It is one of these approaches, intended to reduce (rather than to eliminate) the effort involved in conversion, that will be described first.

Approach

The basic approach is quite simple. All that is really involved are the implementation and usage of a software system that analyzes the binary deck of an IBM 7094 program, converts it to a machine-independent language, and then compiles the machine-independent version for the same or any new computer. Stated in such direct terms, the concept is neither new nor complex.[1,2] However, it did require several years of implementation time.

One of the basic elements in the system is the machine-independent, manufacturer-independent, open-ended self-compiler employed both as the language in which the system is written and as the language into which the IBM 7094 machine-language programs are converted. The language chosen for this role was NELIAC. Because NELIAC has several unique features which contribute importantly to the success of the system, these warrant description.

NELIAC

NELIAC originated at the Navy Electronics Laboratory eight or nine years ago, where it was first implemented on the Univac Countess (AN/USQ 17) computer. After the first 4000 words of the original compiler were written in machine language, they were rewritten in the NELIAC language and compiled. The original 4000 words of machine language were then discarded and all further extensions were programmed in the source language itself. This approach provided a relatively inexpensive method for extending the language to other computers. This can be seen in Figure 1 by noting the way in which the language has branched from one computer to another.

The experience which Lockheed Missiles & Space Company (LMSC) gained with this self-compiler on the IBM 7094 illustrates some of its useful features. Over a period of more than a year, a systems programmer added a new capability to the compiler about once every month. Since he did his programming in source language, each time the programmer added a capability he compiled the new version by means of the old.

The data in Figure 2 show that even though the size of the compiler increased as improvements were added, the total time to compile the compiler was reduced by one-half. This is due to the fact that the compiler was, by itself, a compiled program which could benefit by these improvements. (Obviously, this increase in speed of compilation was not gained at the expense of the efficiency of the object code produced, because the compiler was also object-code produced by the same object code.) The combined effect is shown by the data on Figure 3.

Starting with this compiler on the IBM 7094, the process of converting it to the UNIVAC 1107/1108 involved nine steps as shown individually in Table I.

As can be seen from Table II, the effort required

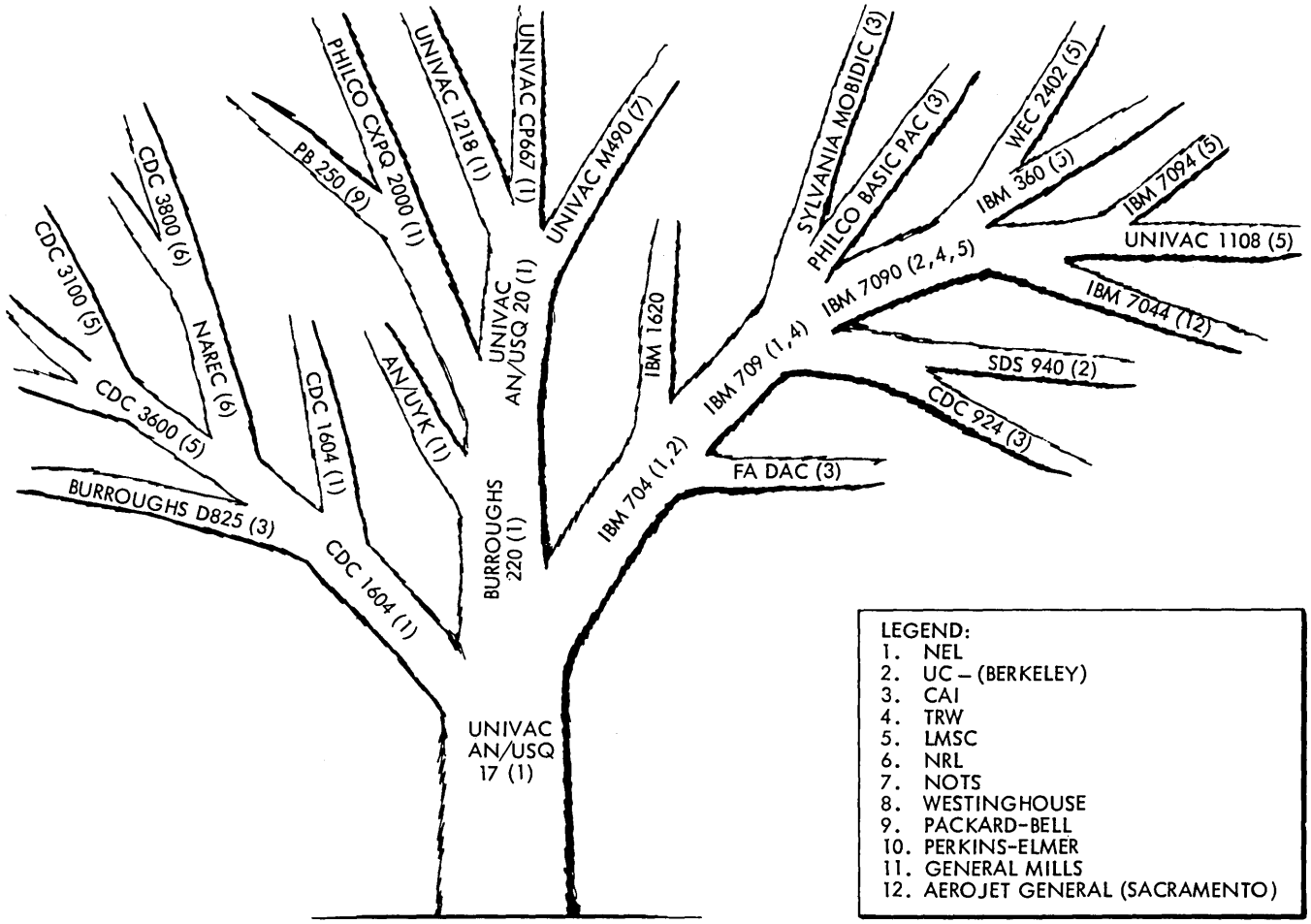


Figure 1—NELIAC tree showing how the compiler on a given computer was employed in the implementation of a similar compiler for different machines

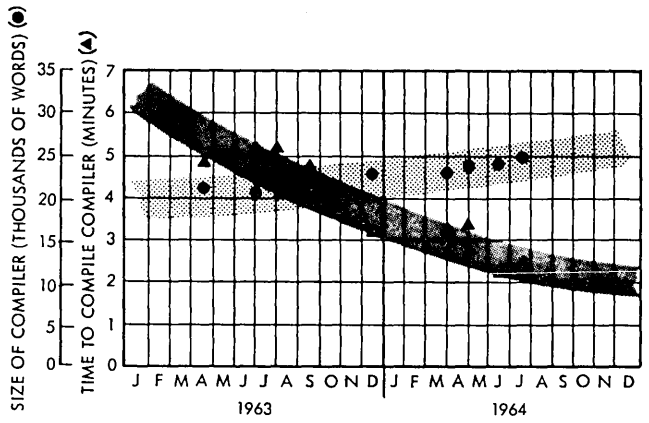


Figure 2—Changes with time in the speed and size of a NELIAC compiler on the IBM 7094

to produce the Intermediate Compiler (which ran on the IBM 7094, accepted source statements, and produced UNIVAC 1107/1108 binary decks) was accomplished in 18 manweeks.

The control of both the language and the compiler which this approach provided was used to make the required extension. While it would have been desirable to use FORTRAN as the language into which to decompile, the limitations of that language were often responsible for the fact that programs were in machine or assembly language in the first place. These, and other limitations noted by Sassaman[3] would have been encountered. Seven of the features which were found necessary to add or have in the language are shown on Table III.

The decompiler was designed to operate upon either an absolute or a relocatable binary deck of a complete program. Theoretically, only a program which is complete and includes all of the subroutines upon which

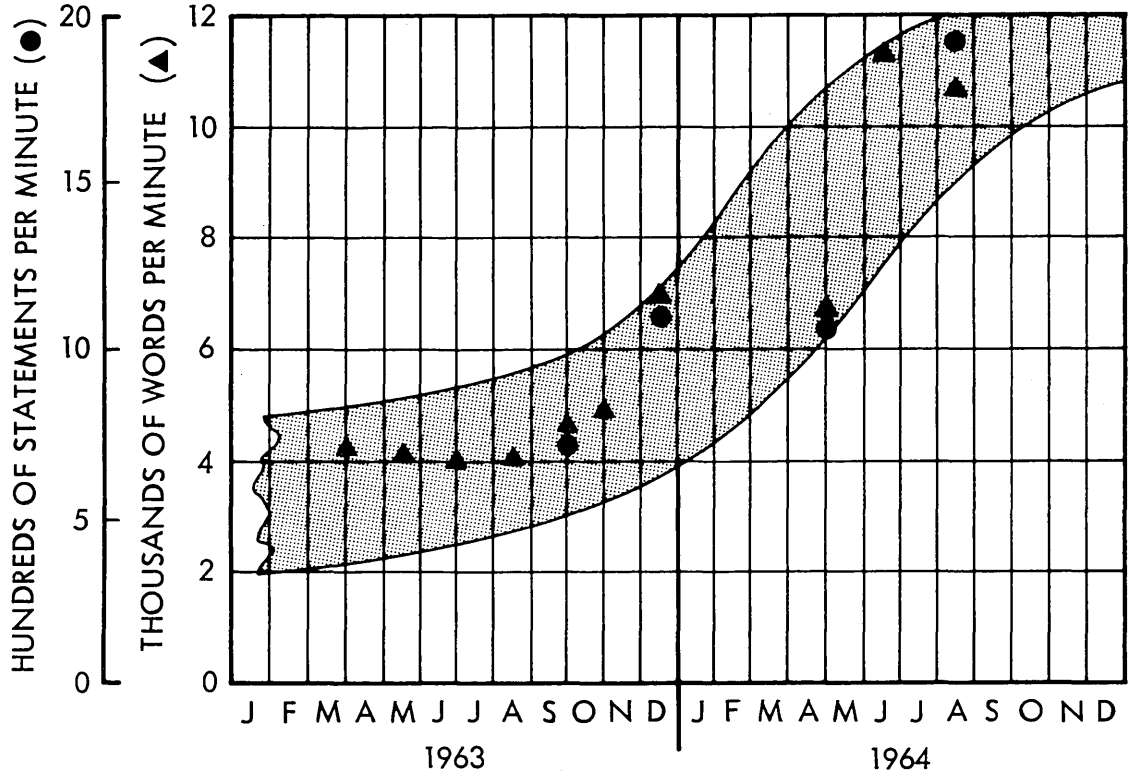


Figure 3—Variation with time in the speed of compilation of a NELIAC compiler on the IBM 7094

Table I

STEPS REQUIRED FOR TRANSFERRING A NELIAC COMPILER FROM THE IBM 7094 TO THE UNIVAC 1107/1108

1. Rewrite generators, in NELIAC, to produce UNIVAC 1107 code instead of IBM 7094 code.
2. Compile the resulting intermediate compiler, using the original IBM 7094 compiler.
3. Compile test case with intermediate compiler on the IBM 7094.
4. Execute test case on the UNIVAC 1107.
5. Rewrite syntactical analyzer portion of the original IBM 7094 compiler.
6. Combine rewritten generators with rewritten syntactical analyzer and compile with the intermediate compiler on the IBM 7094.
7. Load object deck produced in step 6 (on the IBM 7094) into the UNIVAC 1107.
8. Compile and execute test program on UNIVAC 1107.
9. Compile source deck used in step 6 with the compiler loaded into the UNIVAC 1107 in step 7.

Table II

EFFORT EXPENDED IN ACCOMPLISHING STEPS NOTED IN TABLE I

TOTALS THROUGH STEP:	4	8	9
CALENDAR TIME (WEEKS)	9	20	26
MAN MONTHS	4.5	10	12
RUNS (IBM 7094)	47	118	118
COMPUTER HRS (IBM 7094)		12.22	12.22
RUNS (UNIVAC 1107)	2	14	42
COMPUTER HRS (UNIVAC 1107)		0.82	2.50

it calls can be decompiled. In practice, however, this requirement seldom applies because most subroutines do not alter the actual instructions of the calling program.

Again, in theory, the first pass of the decompiler is intended to start with the first executable instruction and determine the entire program flow, thereby separating data from instructions. In practice, as is expected, the line of separation between data and instructions is

Table III

MAJOR EXTENSIONS TO THE NELIAC LANGUAGE REQUIRED FOR THE DECOMPILATION PROCESS

REMAINDERS:	A/B → C∩D,
INDIRECT ADDRESSING:	A[I] + [BL] → C[I],
MODE OF INDIRECT:	A[I] + .[BL] → C[I],
BIT NOTATION:	A(0→14) → B(0→15),
SHIFTING:	A/2 ↑ 5 → A,
INDEXED INDIRECT:	A[I] + .[BL[I]] → C[I],
INDIRECT TRANSFERS:	X. [L+1].

not clear, since an instruction is often altered by storing a new instruction into it. Separating data which are always treated as data from those instructions which are always treated as instructions was, therefore, the simplest part of the task. Determining those cells which served both functions and then converting them to a machine-independent form was the task which required the greatest talent.

Based upon this pass, or passes, the decompiler produces a NELIAC noun list that contains names which it assigns to constants, variables, arrays, switches, etc., together with its determination of fixed, floating or mixtd mode, initial values, etc. The decompiler then generates the NELIAC statement of the logic of the program, insofar as it is able to do so. In this process, it utilizes those extensions noted above where they are applicable and inserts, as crutch coding, any machine instruction which it cannot handle.

Table IV is an example of a printout which shows the absolute octal instructions of a program, together with an assembly listing produced by the decompiler, followed by the program as produced in NELIAC by the decompiler. Table V shows the same items for a subroutine called by the program of Table IV.

Table IV

SAMPLE OF AN ABSOLUTE OCTAL PROGRAM AS READ BY THE DECOMPILER, ACCOMPANIED BY A DISASSEMBLY LISTING AND FOLLOWED BY THE NELIAC STATEMENTS AS PRODUCED BY THE DECOMPILER

```
04013   0774  00  1  00006          AXT   6,1
04014   0500  00  1  04037    E1    CLA   FX5,1
04015   0300  00  1  04045          FAD   FLTPT4,1
04016   0601  00  0  04030          STO   FX7
04017   0500  00  1  04053          CLA   FX3,1
04020   0300  00  1  04061          FAD   FLTPT2,1
04021   0601  00  0  04031          STO   FX6
04022   0074  00  4  04000          TSX   P1,4
04023   0000  00  0  04030          PZE   FX7
04024   0000  00  0  04031          PZE   FX6
04025   0601  00  1  04067          STO   FX1,1
04026   2000  01  1  04014          TIX   E1,1,1
04027   0074  00  4  01000          TSX   EXIT,4
```

```
AC, FLTSP=0.0, FX1(6), FLTPT2(6)=0.0, FX3(6),
FLTPT4(6)=0.0,  FX5(6), FX6,  FX7,  FX8;
START:
   6→I,
 E1:
   FX5[I] + FLTPT4[I] → FX7,  FX3[I] + FLTPT2[I] → FX6,
   P1(FX7, FX6),  AC →  FX1[I],
   I > 1:  I-1 → I,  E1.; EXIT, . .
```

Crutch coding inserted into a program, when in the language of the IBM 7094, serves as a flag to indicate a point at which a conversion expert must study the instruction and supply the proper machine-independent statement. While this expert needs only to determine what the instruction actually does and is not concerned with what the original programmer had intended,

Table V

A SUBROUTINE CALLED BY THE EXAMPLE OF TABLE IV TREATED IN THE SAME WAY

```
04000   0634  00  4  04010    P1    SXA   X4010,4
04001   0560  60  4  00001          LDQ*  1,4
04002   0260  60  4  00001          FMP*  1,4
04003   0601  00  0  04012          STO   FLTPT8
04004   0560  60  4  00002          LDQ*  2,4
04005   0260  60  4  00002          FMP*  2,4
04006   0300  00  0  04012          FAD   FLTPT8
04007   0074  00  4  01000          TSX   SQRT,4
04010   0774  00  4  00000  X4010   AXT   **,4
04011   0020  00  4  00003          TRA   3,4
04012   000000000000         FLTPT8 PZE
```

```
P1;  (D1.D2.)
  {D1*D1 → FLTPT8,  D2*D2 + FLTPT8 → AC, SQRT(AC),} . .
```

it is still true that those things which the decompiler does not handle are often even more complex than the most sophisticated things which it does handle. Consequently, it is rather misleading to mention that the decompiler handles 98 percent of the instructions in straight-forward programs, or more than 90 percent of the instructions in systems programs. While the decompiler does this, the remaining 2 or 8 percent requires a proportionately greater amount of programmer time. Therefore, it seems more reasonable to say that the decompiler eliminates ¾ to ⅞ of the reprogramming work.

While the LMSC Decompiler is not the first to be written, it may be the only one to have reached operational status. As with all software systems, a considerable amount of improvement was required as a result of the first field tests.

Programs converted with the system tend to lose something in efficiency, running only twice as fast on the UNIVAC 1108 as on the IBM 7094 (they should run at 2.4 to 1) and usually requiring ⅓ more core.

While the date available are far from being complete, they do indicate what a very small team of programmers, well versed in both the language and the use of the decompiler, has experienced. They converted some 250,000 instructions of combined FORTRAN II and FAP by "Lifting" the FORTRAN and decompiling the FAP, as shown in the procedure chart of Figure 4. The task required fifteen weeks, with results distributed as shown on Figure 5.

The procedure being employed has not, as of this writing, been extended to an operationally-tested tool for recompiling decompiled programs on the IBM 360 system. While the Tree in Figure 1 shows that a NELIAC compiler has been written for that computer, the extensions noted in Table III have not all been incorporated into it and, consequently, only restricted test cases have successfully demonstrated the process for those computers.

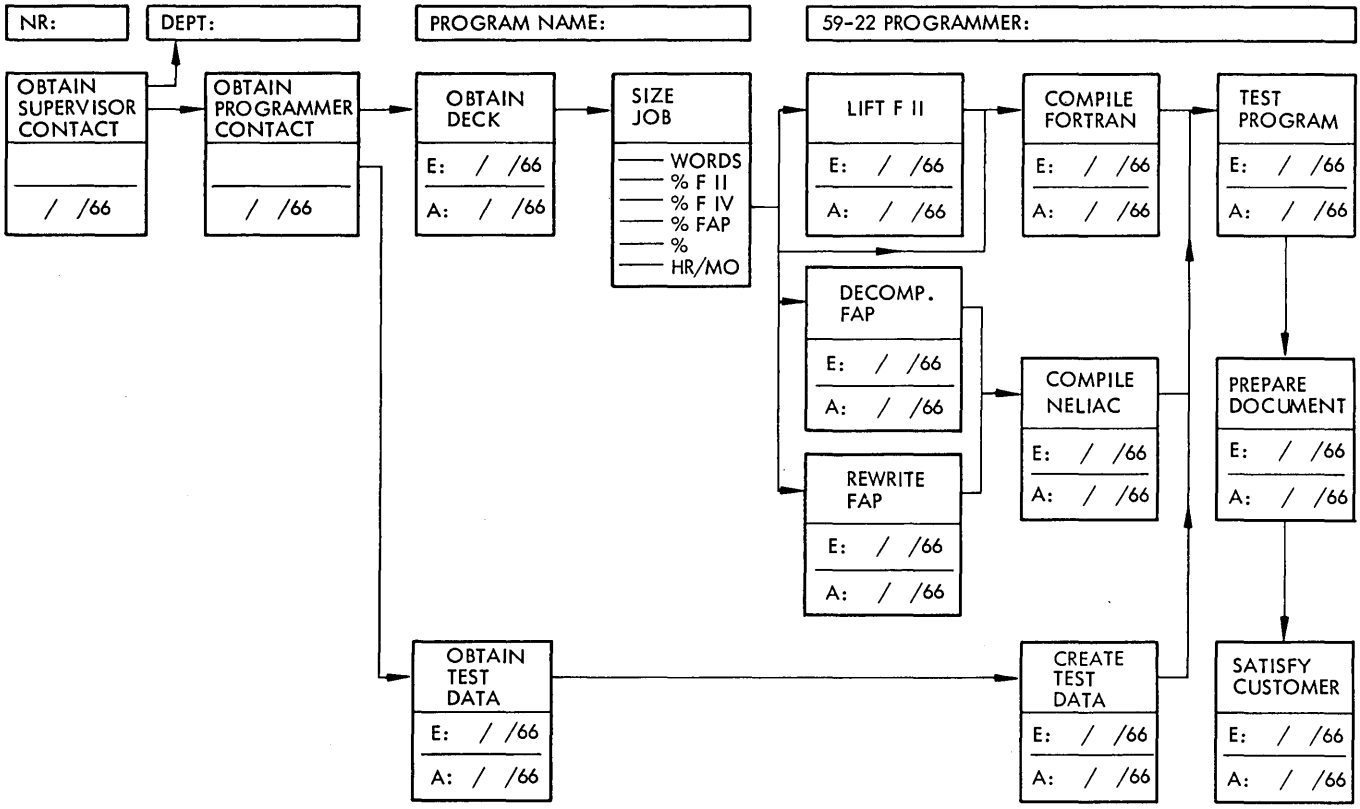


Figure 4—Sample of procedure charts maintained on each of the programs converted

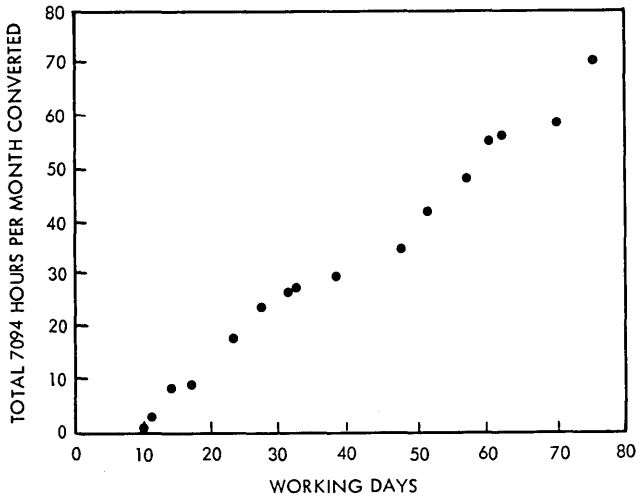


Figure 5—Cumulative totals of computer load (in IBM 7094 hours/month converted by an expert team)

Future strategy

While the methods discussed above are as useful as any we know for handling the conversion of machine-language programs—serving as it does both to convert to the new computers and to "get us out of the box" of machine language—it is not a complete solution for the future.

As pointed out in a paper by Dr. G. A. Garrett,[4] the advent of each new language, or of each new version of an existing language, has been both a source of increased efficiency and a source of additional cost. While the management of a computer center certainly cannot be criticized for wanting to weigh the value of the change against its expected cost, all too often in the past this weighing has been rendered academic. The manufacturers tend to implement only the newest system on their newest machine. While they may provide FORTRAN IV under a given executive system, the newer system will only have FORTRAN V. We must look forward to the possibility that PL/I is only the first term in some finite series, each imbedded in a new and improved operating system. Since the operating systems are becoming, and must become, more and more closely coupled to the operation of a computer program, it will not suffice in the future to have machine-independent computer programs. They must

be operating-system-independent programs as well. This last requirement is worth considerable thought. The number of control cards required to run a program under each new operating system has been rising rapidly, and thus far, at least, there is neither a de facto nor an ASA standard operating system. In fact, it is not at all clear how such a standard could be reached. Therefore, it appears only prudent for the larger users of computers to examine the possibilities of producing their own operating systems.

It is immediately apparent that if these operating systems are to remain as cumbersome and expensive as those now being produced by the largest manufacturers, then no user could afford his own. But as drastic cost reductions in writing compilers followed the introduction of higher-level languages sufficiently powerful to write efficient compilers, it might be expected that the cost of operating systems might also be reduced in a similar manner.

While a fair amount of emotional trauma would be involved because operating systems are the last stronghold of the machine-language programmer, it appears that a properly designed operating system, implemented in an efficient higher-level language, might be transferred to a completely new machine as readily as the compiler described above.

At that point, then, it would be feasible to provide FORTRAN, COBOL, PL/I, and SIMSCRIPT to Base Language translators, each written in the "Systems Language." The Systems Language could, and should, be extended to cover that part of present programs defined as the "Housekeeping" (in David Sayre's terminology, the Phase 2 programming). Then, a single Systems Language to machine-language compiler and an operating system under which they run would be the only programs requiring change to fit a new computer. Furthermore, the languages of the applications programmers would only be modified or extended when the extension was deemed worthy by those responsible for the results, the computer center management.

ACKNOWLEDGMENTS

The author is indebted to the members of the Advanced Software Group of LMSC Sunnyvale who have implemented the system described, and especially to Mr. Robert Stelloh and Mr. William Caudle for providing material for some of the figures presented. Thanks are also due Dr. G. A. Garrett, Director of Information Processing, whose quick perception of its inherent value made the development possible.

REFERENCES

1 A OPLER
Automatic program translation
Datamation vol 9 no 5 pp 45-48 May 1963

2 M H HALSTEAD
Machine-independent computer programming
Spartan Books Washington D C Cf especially chap 11 1962

3 W A SASSAMAN
A computer program to translate machine language into FORTRAN
Proceedings 1966 Spring Joint Computer Conference

4 G A GARRETT
Management problems of an aerospace computer center
Proceedings 1965 Fall Joint Computer Conference

A debate

Main frame memory technology

T.R. FINCH, Chairman
Bell Telephone Laboratories
Murray Hill, New Jersey

Present day digital processing capability has been accelerated into being largely through the system opportunities offered by high-speed, large-capacity, random-access memories. Multi-user, multi-processor, time-sharing systems extend system dependence on improved memory and is bringing forth additional technologies, such as optical and semiconductor to compete with magnetic in the working memory area.

This session presents some of this competition by bringing together magnetic and semiconductor specialists to discuss properties of their technologies related to next generation main-frame memory systems. Magnetics versus semi-conductors; planar magnetic film versus plated wire; and bipolar semiconductor versus MOS will be discussed. The authors represent organizations that have interest and activities in all technologies so they are not asked to represent company or system commitments. Emphasis is placed on memory system requirements and the device-circuit-memory properties of the several technologies may forecast future acceptance and probable use.

Position papers by four of the panelists follow.

Planar magnetic film memories

by Q. W. SIMKINS
International Business Machines Corporation
Essex Junction, Vermont

Planar magnetic film memories offer many advantages for applications as main computer storage units in the capacity range of 200K to 5M bits.

However, before we discuss these advantages, we should define the technological alternatives available. A planar magnetic film memory may involve either integrated drive and sense lines, or separate drive and sense lines on thin dielectric substrates. The film material most generally employed is a nickel-iron permalloy with or without the addition of small quantities of another material such as cobalt. First-order zero magnetostriction is generally sought by composition and deposition parameter control. The film may be continuous, or it may be etched into separate bits on lines. Some form of flux closure is advantageously employed. This may take the form of a ferrite or metal keeper; or, in the case of a fully-integratet structure in which the drive and sense lines are deposited on the same substrate as the magnetic film, complete closure in at least one dimension can be obtained. One such structure is geometrically the equivalent of a flattened wire memory.

The deposition technology employed may be vacuum evaporation, sputtering, plating, or some combination of these techniques. The planar geometry makes all of these techniques practical whereas the cylindrical geometry of the wire memory is somewhat more restrictive. Substrate material may be either dielectric material, such as glass or mica, or it may be metal. Metallic substrates offer considerabl eadvantage in that they may be employed as a reference ground plane simplifying the line geometry and providing low impedance lines with consequent low power and voltage levels.

A wide variety of film array terminal properties are attainable within the definitions given above. Typical properties for a fully integrated coupled film array are given below:[1]

I_{Word}	200 mA (6 nsec use time)
I_{Bit}	15mA
V_{Signal}	1-2 mV
Word lines	7 mils on 14 mil centers
Bit lines	4 mils on 6 mil centers

Somewhat lower packing density and higher bit current for equivalent signal level are required in a non-integrated array.

Advantages

1. High-speed operation

Thin magnetic films switch very rapidly. Thus, the access and cycle time of film memories is dependent not on the individual device properties but rather on the

[1]Hsu Chang, "Coupled Film Memory Elements," J. Appl. Physic vol. 38, no. 3, p. 1203 (1967).

array parameters and circuit and packaging designs employed. Planar film memories have relatively low drive currents, low impedances, and, consequently, low power. This requirement permits the use of very high-speed circuits. In addition, array packing densities of the order of 10^4 bits/in.^{2} are achievable, leading to relatively short drive and sense lines with relatively large numbers of bits per line. When properly terminated, these transmission lines will result in fast array recovery. This combination makes possible the design of film memories of a million bits or more with cycle times of <100 nsec.

2. Low power

Word currents in the 100-200 mA range are attainable as are bit currents in the 10-30 mA range. Line impedances of substantially less than 50 ohms are easily realized. The low power requirement permits the use of very high-speed circuits and also provided compatibility with monolithic drive and sensing circuits thus contributing to low cost.

3. High packing density

Array packing densities of the order of 10^4 bits/in.^{2} or even higher are achievable. These are at least an order of magnitude better than can be expected with wire memories.

4. Large-scale integration

As technological advances are made, the planar technology permits us to pack more and more bits on a single substrate thus reducing the interconnection problem and simplifying the total memory packaging job. This integration will reflect in the long run on product cost and product reliability.

5. Magnetostriction effects

The use of a rigid planar substrate reduces the sensitivity of the film structure to magnetostrictive effects.

Disadvantages

1. Complex processing

The highly integrated structure requires somewhat greater complexity in processing. However, this is an initial entry barrier: the compensating advantage of simplified packaging for the entire memory and the reduced number of interconnections will far outweigh this disadvantage.

2. Low signal levels

Although low signal levels are not necessarily inherent to the planar structures, they are the natural consequence of attempts to reduce power level and increase density. Indeed, they represent a limitation in the memory design and will require design of high gain/low noise sense amplifiers. On the other hand, the electrically generated noise level in the planar array is very low, particularly if a metallic ground plane is used; and, as a consequence, the inherent signal-to-noise ratio that may be obtained is still quite favorable.

Plated magnetic cylindrical thin film main memory systems

by G. A. FEDDE
Sperry Rand Corporation
Univac Division
Philadelphia, Pennsylvania

INTRODUCTION

Since research and development work started on permalloy thin films for random access memory elements in the mid-1950's, the rapid development of ferrite core memory technology has prevented the general use of magnetic thin films. This occurred because magnetic film memories did not have enough advantage in the cost-performance-capacity comparison with ferrite cores.

For the next generation computer main memory, it appears that magnetic film memory elements will be dominant compared to ferrite cores. The next generation main memory is defined to be within the limit given below:

Cycle time — less than 500 nanoseconds
Capacity — larger than 2×10^5 bits
— smaller than 5×10^6 bits
Random access to any word or byte

Based on film geometry, there are two main forms of magnetic film memory elements—planar and cylindrical. Based on manufacturing technology, there are two principal methods of film deposition—vacuum evaporation/sputtering and electroplating. For the business-commercial environment, the cylindrical magnetic thin film appears to offer the best combination of advantages. For cylindrical films, electroplating is the most practical means of depositing the film. Therefore, for the purposes of this debate, it is asserted that plated cylindrical magnetic thin film memory elements will become dominant in the next generation computer main memories. The degree of correctness of this assertion should be apparent within the next two to five years. If the assertion is essentially correct, it is because of the factors discussed in the following paragraphs. It must also be recognized, however, that even if all but one major manufacturer committed their next generation main memory designs to cylindrical films, they could

remain subordinate if millions of bits produced per year is the only measure of the degree of dominance. The converse is also true. Whatever memory element and technique is successfully used, by this same major manufacturer for their "next generation main memories", could be dominant based on production volume. An equally signiicant measure of dominance, from the standpoint of technical performance-cost factors, is the percent of computer manufacturers that commit their new generation designs to plated cylindrical magnetic film memory elements.

Elemental advantages

Practically all widely used magnetic devices have closed flux paths except for those devices in which an air gap is absolutely essential to their functioning. Magnets used in motors. voice coils, and kitchen cabinets are examlpes in which an air gap is essential. Air gaps in magnetic memory elements are generally nuisances, or worse, and there appears to be a strong trend in planar film memory devices to close the gap. A single plating operation on a continuous cylindrical substrate should result in a significant advantage for plated wire compared to the alternate multiple process steps required in planar film technology to achieve a closed flux path.

Another advantage is realized from the relatively low cost of the capital equipment and high degree of mechanization possible in the large volume manufacture of plated cylindrical thin film memory elements.

The continuous substrate provides another advantage in the production testing and assembling of the memory stack. Any memory element used in a nonseparable group of similar elements puts a severe demand upon process control and uniformity. This is true for both cylindrical and planar magnetic films. Consider the tyranny of numbers over yield using a wire length or substrate area for 256 bits. A ferrite core yield in the pressing sintering process of 95% is excellent. The probability of achieving 256 consecutive good cores at this yield is 1.98×10^{-6}. This is, of course, totally unacceptable for a substrate yield. The wire substrate is tested before cutting, so that when a bad bit is detected somewhere within the desired wire length (256 bits long), the wire can be cut and the short segment discarded or used in a different memory design utilizing shorter length wire. This compares to the testing of planar films after the sense line has been bonded (open flux path) or deposited (closed flux path) on the substrate and only the word lines need to be added. At such a stage in planar film technology, a detected bad bit rejects the entire 255 good bits instead of a fraction of that number. If we assume a manufacturing run of 5.12×10^6 bits total; an individual bit yield of 99.728%; a desired substrate size of 2.56×10^2 bits; and defects are randomly distributed; then we find that cutting at a fixed size (256 bits) gives 10^4 length with all bits good, and 10^4 lengths with one or more defective bits. If we cut the substrate whenever a defective bit is found, the same total production yields 1.3×10^4 lengths with all bits good. This 30% improvement in manufacturing yield provides an important advantage for the continuous substrate manufactured and tested plated wire memory element.

System advantages

Perhaps the most significant system advantage available to users of plated magnetic cylindrical thin film memory elements is a non-destructive readout capability. For main memory use, NDRO with equal Read-Write drive currents is most advantageous. It allows the greatest possible flexibility of organization and operation. For maximum economy, many memory words or bytes) may be accessed by a single word drive line without need for more than one set of sense amplifiers and bit current drivers. The set contains only the number of amplifiers needed to process the bits of one word (or byte) in parallel.

NDRO provides a higher average speed memory memory system. If we normalize cycle time to the time interval from initiation of a write cycle to the earliest possible initiation of a read cycle within the same module, then successive read cycles or a write cycle following a read cycle may occur at a 50% to 60% cycle time. For a system using 3 read cycles for each write cycle, on the average, an overall average cycle time of 62.5% to 70% can be achieved if the computer system is organized to take advantage of this fact.

The nominal value of bit-write current, compared to some planar film memories, makes integration of the bit dimension electronics straight-forward. At the present time, plated magnetic cylindrical thin film memories in production require 600 to 900 milliampere turns of word drive. This is not a fundamental limitation but indicates that the drive current amplitude has not been a major drawback and other aspects of the development of these memory elements has received priority. The same techniques used by some planar film memories in the development laboratory, such as thinner films, lower H_k, and smaller geometries, are equally applicable to cylindrical films to achieve low values of word drive current if necessary.

Permalloy thin films, whether planar or cylindrical, have high curie temperatures and are, therefore, relatively insensitive to changes in the temperature of their

environment. In addition, both geometries switch a small volume of magnetic material so that drive line inductance and impedance are determined largely by the detailed physical construction, particularly if a magnetic keeper is used. From a practical standpoint, the necessary drive voltage for a cylindrical film memory, with 640 bits per word drive line and 40 ns. rise time, is only **5** volts which is easily handled in a variety of ways. These considerations lead to a standoff between cylindrical and planar films. A similar situation exists regarding operating current margins. Cylindrical film NDRO memories can be operated with $\pm 20\%$ margins on word and bit currents if all the wires in a stack come from well controlled production lots. Since these margins are larger than necessary, wider variation in lot-to-lot parameters are permitted to improve manufacturing yield. Thus typical NDRO cylindrical film memory operating margins will be $\pm 7\%$. For a fair comparison to planar film DRO margins which may be a little wider, it must be remembered that DRO operation of plated wire memories with the same wire population, would allow margins of at least $\pm 10\%$.

Rather than attempt to quote price per bit and its relationship to speed and capacity, the following table shows relative cost only. Price is influenced by many non-technical factors and can be misleading. The best reference point that can be given to relate cylindrical film memory cost to absolute numbers is that 8192 and 16384 byte plated wire NDRO memories (9 bits per byte), with a 300 ns. access time and 600 ns. cycle time, are significantly lower cost for us than ferrite core memories of identical size and speed but with DRO.

Relative Cost	Nanoseconds Cycle Time	Module Capacity (bits)
1/2	1000	1×10^7
2-3	300-400	1×10^6
6-8	100	2×10^5

The closed flux path geometry provides a substantial construction advantage. The plated wire is tolerant of wide variations in the relative alignment of word drive lines to the longitudinal wire axis which is also the "hard axis." This kind of mis-alignment does not produce a favored direction of information storage since there is not a component of word drive field parallel to the circumferential easy axis. Plating the magnetic film directly on its own sense line also provides constructional simplicity since no other drive or sense line is used. The use of a continuous film eliminates the need for any lengthwise alignment of wire to word lines and allows the center line spacing tolerance to be determined by other factors. Finally, the construction of a plated wire memory stack makes it very easy and quick, (approximately 1 minute), to replace a wire that has slipped by the less than perfect element testing level preceding stack assembly. These advantages combine to make the fabrication and assembly of a cylindrical film memory appear to be substantially simpler than a ferrite core memory.

SUMMARY

The foregoing very brief discussion has spotlighted the basic advantages that are expected to bring plated magnetic cylindrical thin film memory elements to a dominant position in the next generation computer main memory. They all point to higher performance at lower cost than present main memories.

The case for bipolar semiconductor memories

by R. S. DUNN
Texas Instruments Incorporated
Dallas, Texas

The discretionary wiring approach to the large scale integration of active memories[1] permits several thousand memory cells to be interconnected on a single substrate. It also allows drivers decoding, and sensing to be integrated with the memory matrix. This feature is vitally important in reducing the number of array connections required, maximizing packing density and widening noise margins. Such a bipolar memory array with 1600 active cells is shown in Figure 1. This technology is also appropriate for MOSFET memories. Some method of minimizing the ratio of the number of cells to the number of connections is essential to realize the MOSFET potential for extremely high packing density. However, integration of the MOST control circuitry is only achieved at a substantial speed and size penalty. Thus MOSFET discretionary wired LSI memories will compete most effectively at the slower speed end of the memory spectrum. Multi-chip assembly technology in particular beam lead systems[2] are attractive memory techniques since bipolar control circuits and MOS storage cells may be intermixed. However, the assembly costs and reliability are not likely to be competitive with the bipolar LSI array. The state of the art bipolar memory array such as that shown in Figure 1 is fabricated by interconnecting 16 bit memory circuits with a second level of interconnection to form a larger function. The most important ad-

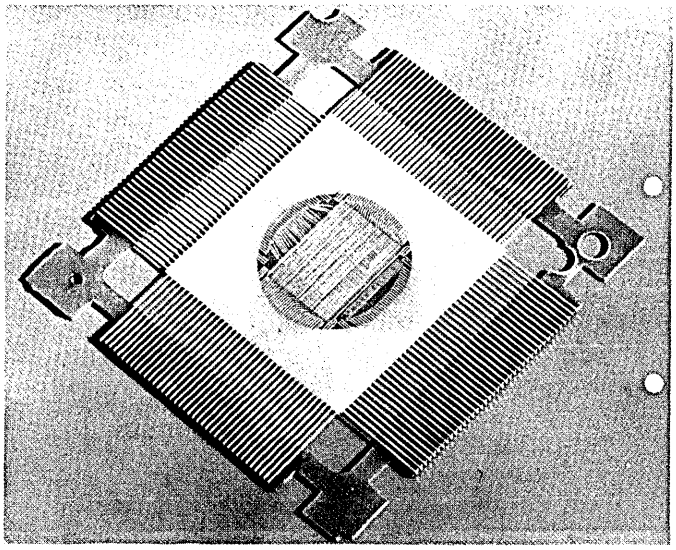


Figure 1—Active memory array

vances required to produce this array have been:

1. High yield circuits—extended through the use of redundancy
2. Accurate automatic test
3. Efficient computation of the discretionary mask
4. Low cost mask generation
5. Very high quality multilevel technology

The cell using 0.0003″ mask dimensions occupies an area 0.009″ x 0.015″, as shown in Figure 2, and has a read delay of 15 nsec and typical slice yield in excess of 80%. Projections on area per bit will be limited by

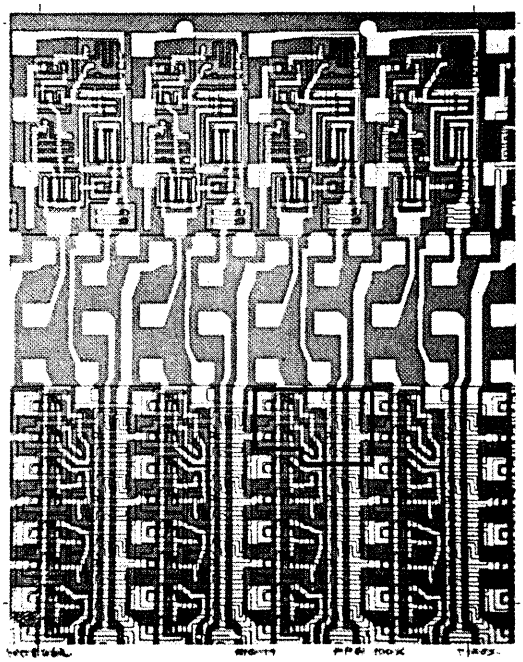


Figure 2—Active memory circuits

current distribution problems more than mask resolution capability and only modest reductions to 0.006″ x 0.01″ are forecast over the next two years. However, 60 square mil memory cells provide over 8000 cells on an 1¼″ slice as shown in Figure 3. The discretionary wiring approach will then allow products to be made

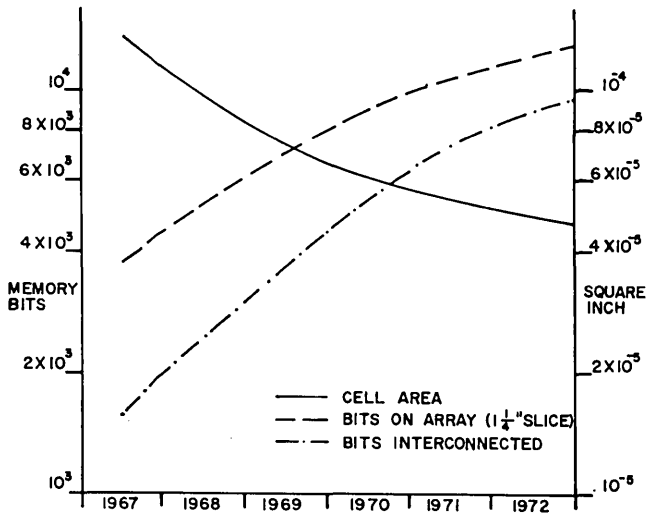


Figure 3—Bipolar LSI array trends

with over 5000 of these cells interconnected and integrated with drive and sense circuitry. This then is more than the matrix or stack, it is a functional memory module which may be extended by the addition of more modules, and represents the bulk of the memory system cost. Relative to a fixed interconnection bar of 64 bits of similar cell quality with a 5% to 10% yield per slice the discretionary connection will be 5 to 10 times more efficient in the use of the cells on a slice. A 5000 bit array price of $100-$150 would yield a per bit cost of two-three cents and will be achievable within the next few years. Further advances using larger slices will further reduce bit costs by larger integration up to 10,000 bits per slice and represents the most significant incentive for the use of larger slices. A block diagram of a 128 word by 36 bit array is shown in Figure 4. With address decoding less than 100 signal connections to the array are required.

The LST bipolar memory is hence applicable to memory functions from 5000 bits onwards. The next level of modularity beyond a single array will be several arrays on a circuit board. A board of 1024 words or eight arrays represents a convenient size for wired OR data line connections. In general the flat board assembly is preferable in terms of maintenance and thermal requirements to other assembly techniques.

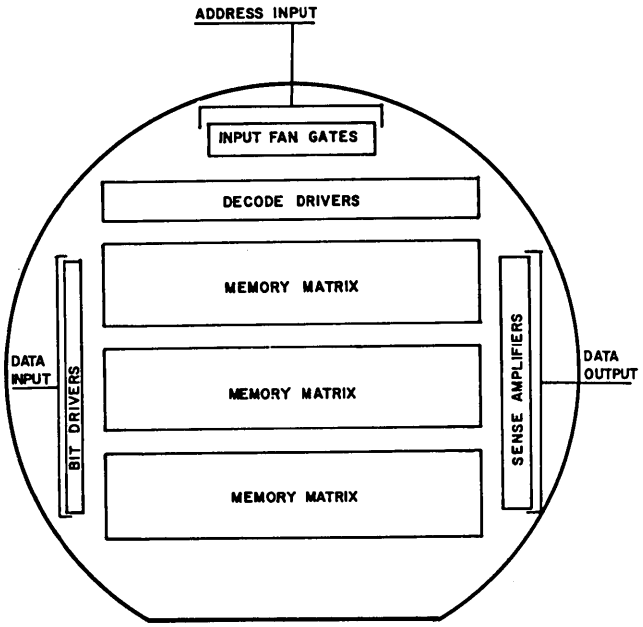


Figure 4—Memory array organization incorporating driver and control circuitry

The access through the 1024 word memory board will be kept below 90 ns with a dissipation of around 30 watts, i.e., 1 mW/bit.

This speed power performance requires only modest advances from today's arrays. The board module size is convenient for small memory applications and indicates the method whereby LSI memories will establish the production volume and the impetus for main frame memory applications. The LSI memory being produced for the Air Force by Texas Instruments Incorporated falls into this category.

It is not anticipated that bipolar main frame active memories will find commercial use prior to 1970 when 5000 bit arrays are available in volume. Some advantages of this class of memory are a common assembly technique. One cooling system and a single impedance level and signal level may be used throughout as only one technology is used for storage and for control. Ground return techniques and noise considerations are also simplified, due in part to the high degree of modularity. Power supply needs are uniform and the number of supplies minimized. It is not, however, expected that any significant advances in volume or dissipation will be achieved by active memories. An examination of the speed and capacity of such a memory and price trends indicates no real reduction for memory systems over 500,000 bits. Target figures are 4-5 cents per bit for memories with an access time of 100-150 nsec by 1971. The larger memory capacity needs will be met by the duplication of whole memories. This organization, coupled with NDRO operation and pipelining information flow cycling through the memories presents many opportunities for increasing the efficiency of memory utilization. The opportunities in active memories to the semiconductor manufacturer are such that this area will be one of the major markets within the next five years.

REFERENCES

1 R S DUNN G JEANSONNE
Active memory design using discretionary wiring for LSI
ISSCC 1967

2 J E IVERSEN et al
New implementation of bipolar semiconductor memory
ISSCC 1967

Magnetics—still the best choice for computer main memory

by RICHARD J. PETSCHAUER
Fabri-Tek Incorporated
Edina, Minnesota

In virtually all of the computers designed in the last 10 years, static (non-rotating) magnetic devices have been used for the random access main memory. Toroidal ferrite cores have been the predominant element used principally because of their relatively low cost and the circuit economies resulting from coincident-current operation. Batch-fabricated magnetic structures have received increased attention during the last 5 years by workers in the field because of their promise to lower memory array cost and increase speed. Most of the activity has centered around the use of metallic films in the form of planar structures or cylindrical surfaces as in the plated wire. Commercial exploitation of these film memories has begun as evidenced by recent announcements in the industry. Present applications are in the area of high performance memories, but there is reason to expect eventual lower cost as well.

Role of semiconductor memories

The rapid and widespread use of integrated circuit logic devices by computer designers, coupled with further improvements in semi-conductor technology has raised the question of the impact of Large Scale Integration (LSI) on computer equipment. It is generally agreed that this is a very complex problem. The use of Large Scale Arrays for logic require solutions to the problems such as forming interconnections, debugging logic networks, specifying and testing multi-

state arrays, and attempting to standardize arrays so that reasonable production runs and low per unit design costs can be obtained.

Since most of these problems are eliminated for the memory function, where repetitive cells can be used, some attention has been directed toward implementing memory with integrated semi-conductor arrays. However in a memory of sufficient size for main storage (5×10^5 - 5×10^6 bits) several other serious problems arise which the writer feels will prevent their practical use in the foreseeable future. The principal of these will be excessive storage cell cost considering the competition of a magnetic memory. Comparison must not be with present logic gates of about 50c, but with a magnetic array of about 1c per bit or less. In an attempt to reduce cell cost, density must be increased substantially aggravating the processing, interconnection, and yield problems. Another serious problem is the failure rate of equipment with such a large number of semi-conductor junctions, crossovers and connections. Based on present integrated circuit failure rates, a 10^6 bit memory would have a mean-time between failure of only a few hours. Several orders of magnitude of improvement are needed. Even if this could be obtained, accessability for testing and replacement of defective memory cells must be considered and this could complicate the design. Other secondary problems which must be solved with I-C memories are handling the volatility problem, keeping cell dissipation low, and removing heat from the silicon.

Fruitful exploitation of I-C memories need not be for main memory. They offer some obvious advantages for high speed scratch pad memories, shift registers, small associative memories, some read only memories, and other general areas requiring a high ratio of logic-to-storage functions. The interest in main memories seems to be less with the semi-conductor workers than it is with possible users. The chief advantage of I-C memories is their compatibility with logic. Their relatively low cost of selection circuitry and their high storage cell cost causes cell cost to predominate at capacities above 10^3 - 10^4 bits. Per bit cost of magnetic memories continues to drop as capacity is increased to 10^5 - 10^7 bits. It is not expected that the cost-crossover point will exceed 10^4 - 10^5 bits. Both types of memories will probably exist together in new system designs utilizing some distributed semi-conductor logic and memory together with a magnetic main memory providing the bulk of the storage function. The magnetic memory could, of course, be made in modules to increase throughput or allow overlapping or simultaneous operation.

Benefits of magnetics

The main advantage of magnetic memories is probably that they were invented first and are now in a fairly advanced state of perfection. This applies particularly to the ferrite core. It must be remembered that the development of any new memory approach is less of a design problem and more of the development of a manufacturing process. This takes time and money. Much of this time has already passed for magnetic memories and a good part of the money has been recouped during a period when the lack of a competing technology allowed start up and learning costs to be charged off.

At the same time it is felt that there is considerable possibility for improvement in the cost-performance of magnetic memories. Primary reasons for this are the relatively simple and reliable magnetic array which can be used, the fact that is amenable to batch-fabrication, as well as the practicality of integrating substantial parts of the electronic circuitry.

Future trends in magnetic memories

Trends which we should expect to see in future magnetic memories are listed below:

1. Trend toward simple cell structures—2 or 3 wire arrays
2. More automated assembly and conductor termination or batch-fabricated arrays.
3. More fully automated plane testing
4. More standardization
5. Extended use of integrated or hybrid circuits
6. Improvde methods of packaging for stack and stack interface circuits to reduce packaging and assembly costs.
7. Reduced physical size

Some examples of possible magnetic memory approaches for future main memories are listed below. Basic manufacturing costs do not include development costs, return on investment, administrative or selling costs.

Examples of possible magnetic memory approaches

1) Ferrite Core Memory
   3 wire, 2½D 14/18 mil core; 500 nsec cycle time; 10^6 bits, hybrid-integrated Y (digit) drivers and diode select; 2-3¢ basic manufacturing cost.
2) Planar Film or Plated Wire Memory
   2 wire, 2D; 2 x 10^6 bits (16K x 144), 250 nsec - 500 nsec cycle time; hybrid-integrated sense - digit circuits, discrete word selection; 2-4¢ per bit basic manufacturing cost.

3) Advanced Ferrite Core Memory
   2 wire, 2½D, 16/11 mil core; 500 nsec to 1 usec cycle time; 5 x 10^6 bits;
   1¢ per bit basic manufacturing cost.

4) Second Generation Planar Film/Plated Wire Memory
   2 wire, 2D, higher density stack; improve flux closure 4 x 10^6 bits, (32K x 144); 200-400 nsec cycle time, integrated sense digit circuits and word selection; 1c per bit basic manufacturing cost.

SUMMARY

Integrated circuit memories, while finding use for small capacity stores, are not expected to replace magnetic memories in capacities as great as those needed for main memory (above 10^4 - 10^5 bits) primarily because of cost, manufacturing problems, and reliability. We can probably expect a further increase in the types of memory approaches employed — another level in the heirarchy of memories. The system designers will probably have the final answer in determining the proper role and relationship of small fast semi-conductor memories and the lower cost magnetic stores.

Bulk core in a 360/67 time-sharing system

by HUGH C. LAUER
Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

In the fall of 1965, Carnegie Institute of Technology decided to install Large Capacity Core Storage (LCS) as the auxiliary storage device on its IBM 360/67 Time-Sharing computer system. The bulk core will be used as a swapping device, replacing the drums of conventional configurations, and as an extension of main core memory. The decision was motivated by an analysis which yielded the following results:

- The effective rate at which the system can deliver pages to user tasks is increased to its theoretical limit with LCS, representing a significant improvement over drum performance.
- The potential response time to users is decreased because LCS has no rotational delay.
- Less main core is needed for effective system operation.

In addition, LCS provides the memory necessary to support specialized computing requirements such as artificial intelligence research or large table-driven compilers.

In this paper, we will present the assumptions and analysis which shaped the configuration, look at a model of a drum-oriented system, compare it to LCS, and present the Carnegie system and implementation plan.

Background and assumptions

When a task executes in machines of the class of the IBM 360/67 or GE 645, only the local portion or *neighborhood* of its program and data which is actually relevant at a given moment need be in core. What the program "sees" is not physical core, but a space of addresses called a *virtual memory*. A hardware mechanism translates each address into a core location, if that address is part of the neighborhood in core, or to a supervisor interrupt if it is not. In the latter case, the monitor causes the block of virtual memory, called a *page*, which contains that address to be read from a swapping device into core. Thus, a machine with a small physical memory can allow several large programs to share it simultaneously. The user is absolved from preparing overlays and deliberately fragmenting his program, while the system bears the burden of swapping pages in a timely and efficient manner. It is the efficiency of this swapping process that concerns us here.

There are some special restrictions on the user program in this system. For example, it is unwise to refer randomly to locations in virtual memory; that would generate a very high demand rate for pages. Rather, the task should work in reasonably small neighborhoods for relatively long periods of time. The system is, however, a step toward the goal of providing a comprehensive time-sharing system in which users can operate conversationally or non-conversationally, without restrictions on language, facilities, or program structure. While we are realistic enough to see that this goal will not be closely approximated for several years, we do allow it to affect our thinking.

What we have just described is a system which simulates a large core memory with on-line storage, usually a disk or drum. The time-sharing capabilities come with the structure of that memory, not its size. It is worthwhile then to look at a large core simulation system. One successful machine, at the University of Grenoble, provides an 800,000 LISP word memory on a 7044 with disk.[1] In this system, a swapped program ran approximately one-third as fast as the same program residing in core. However, from the published data, it can be seen that the mean time between calls for new pages (one might call it the *mean free path* of the program) is a function of the number of pages already in core, and is usually greater than three seconds. If it is smaller, then the swapping overhead time becomes large relative to the user task time, and the system loses efficiency.

Clearly, the mean free path depends on the structure of the programs in the system. Investigators at SDC have examined the behavior of several programs that might be typical in a time-sharing environment.[2]

Existing jobs from the Q-32 machine were run under a simulated paging system, and the mean free paths were measured as a function of the number pages in core. It was discovered that with neighborhoods of only a few pages, the mean free path was extremely small and the demand rate for pages per unit of task CP time was high. In Figure 1, the number of pages

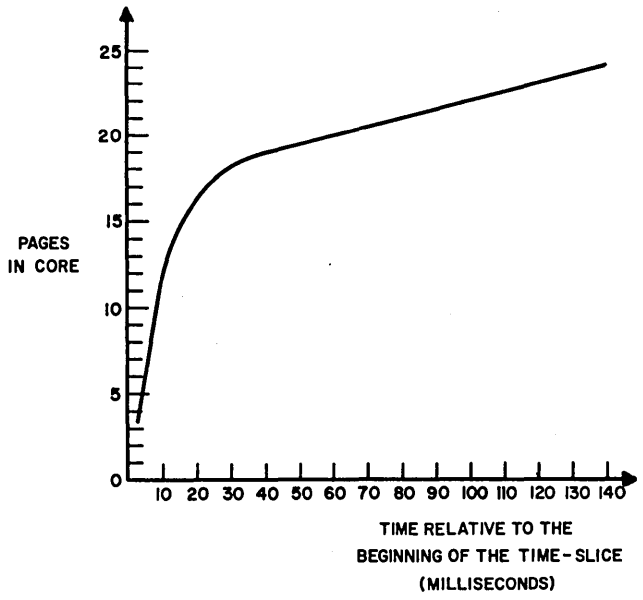


Figure 1 – Reproduction of data presented by Fine, et al SDC

in core is plotted against the time within a time slice, assuming that the task starts out with no pages and each is brought in on demand. It can be seen that once a *working set* of pages is available, the demand rate goes down from its initial high level. However, the authors point out that with the program structure as it is, the paging mechanism is bound to congest very quickly, and "there would be little chance of processing-fetching overlap." Paging demands would overwhelm the system.

We must be careful about translating such experiences to other computers, for the environment has a heavy impact on the character and structure of programs. For example, in the Q-32 analysis, they were 30-46 pages long. In the 7040 and 7090 systems programs are less than 32 pages long; yet results for these machines strongly influenced the design of the 360/67 and its software.

In the IBM Time-Sharing System/360 (TSS), there is a tendency to code in many small modules, to separate data from procedure, to write re-entrant routines, and to functionally fragment programs. None of this appeared in earlier machines, and all of it tends to increase paging demands. Early experiences at Lincoln Laboratory and at IBM with TSS indicate that estimates such as Figure 1 are conservative, and that the system is more page-bound than was anticipated. In particular, the knee of the curve lies at about 40 pages. The Carnegie LCS-oriented system is an attempt to create a machine which is not page-bound while minimizing restrictions on the structure of task programs.

In the analysis that follows, we must make some assumptions and avoid others, based on experience and "what's reasonable." In particular:

- We assume that the system is page-bound, with a demand rate of the order of hundreds of pages per second. Specifically, we assume that the amount of swapping channel time is greater than the amount of CP time which it supports, and we direct our efforts to eliminating the consequent CP idleness at a reasonable cost.
- Our analysis will be oriented about the stochastic nature of the system. We assume that each task, and all random variables describing it, are independent of all other tasks in the system.
- We are unwilling to assume that any random variables have particular distribution functions, with only the following exceptions:
  1) The wait-time between the moment a task requests a page to be read and the time at which the swapping device becomes available to service it is uniformly distributed over the length of the swapping channel program.
  2) The variable designating the location of a page or group of pages on the swapping device will have either the uniform or singular distribution.

  Both are justified because the state of the art does not now provide mechanisms for decreasing the expected wait-time to find pages in this environment. Any such techniques depend inherently upon program structures, about which we know very little.
- We ignore file activity and concentrate only on the swapping process. A more comprehensive analysis, which will extend these results to account for other devices, will be published in the future.
- We will ignore correlations between the state of the system at the end of a time slice of a task and the state of the beginning of the next time slice for that same task. We can treat shared pages, which are swapped only infrequently, if at all, as part of the resident system, and not as part of the swapping load. The demand for space for non-shared pages will be sufficiently high that only

a negligible number of them will survive in core during the interval between time slices.

- After we have experience with the system, we will take advantage of correlations between system states to improve scheduling algorithms; now we can only do this in a limited way.
- Inefficiencies in monitor coding can be ignored. We are interested in the hardware capabilities, not software performance, so we will assume the ability to write clean code. The supervisor overhead for an interrupt can be assigned to the task which caused it. Consequently, the *task time* is the total amount of CP time dedicated to that task for both user computing and overhead, i.e., the marginal increment of computing load added to the system by that task. The fact that overhead may amount ot 1% or 90% of the task time is irrelevant to us here.

With these assumptions, we can proceed with the analysis.

Drum-oriented system

The drum which we consider as a swapping device has one read-write head for each track, but only one head can be connected to the channel at any moment. There are p pages recorded on the circumference of each track, and there is sufficient space between pages to permit head switching. Thus, it can read the first page on track a, followed by the second page on track b, and so on through the p^{th} page on track x, all in one revolution. We can regard the drum then as a sequence of *p slots* passing by a single head. When the head is over a particular slot, one and only one page may be read or written in that slot. The drum operates asynchronously from the CP under the control of its own *channel program*. Once an operation has been started, it may not be interrupted or altered except in the case of malfunctions.

Figure 2 will help describe the system which we model. At time A, a channel program was initiated to run until time B. Tasks 1, 2, 3, 4, and 5 execute in turn until each develops a page request. By time B, all requests have been analyzed and a new channel program has been prepared. The figure shows examples of what can happen:

1) The page for task 4 was located on the first slot, so that task 4 is ready to execute almost immediately.
2) Task 1 is serviced next, then task 2.
3) The page request for task 3 happened to fall in the same slot as that for task 2, so it must wait a full revolution before it can be serviced.
4) Task 5 has a long wait because of the location of its page on the drum.

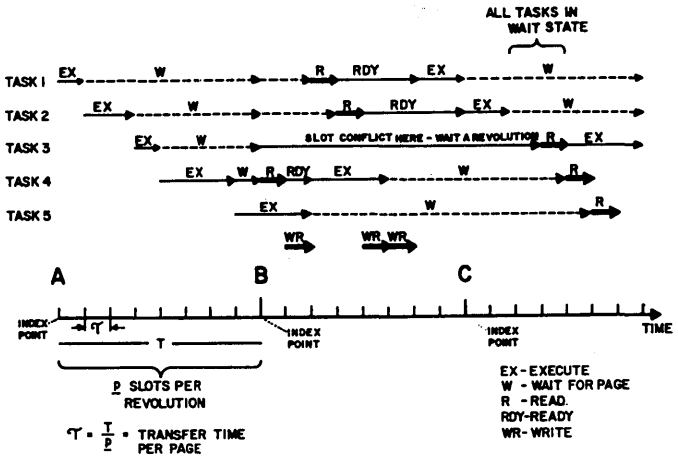


Figure 2 – Example of system execution

5) Several slots which are not needed to service tasks at the moment are used to write out old pages from core in order to make room for new pages.
6) Shortly after time C, the CP enters the wait state because all of the tasks are waiting for pages, and there are no others ready.

Though we had four requests at point B, we could only service three of them because of a slot-conflict. Of course, had the system been able to anticipate either the slot conflict or the idle time, it would have taken corrective action; but our assumptions about the independence of tasks and the randomness of page locations preclude this.

In the following paragraphs, we will calculate the rate at which the drum can deliver pages and observe the relevance of this to the mean free path of tasks. Then we will discover a *lock-out* phenomenon, by which the drum causes pages to be withdrawn from usable core. We will also explore some ideas to swap groups of pages as a means to overcome some of the problems we encounter.

Deliverable page rate – demand paging

We consider the case in which all channel programs are the length of one drum cycle, of time T. The expected rate at which pages are read is a function of k, the number of tasks in the drum queue, and f, the probability that a page must be written out to make room. It is possible that $f < 1$, since some pages in core may have valid copies on the drum already (by virtue of being re-entrant, or otherwise not changed since being read). Then for each page requested, the swapping channel must handle an average of $1 + f$ pages.

Thus, the maximum average request rate which a p-slot drum can service is $\frac{p}{1+f}$ per revolution or $\frac{p}{(1+f)T}$ pages per second. The one-sided effect which occurs when f is greater than its mean is negligible if the system maintains a buffer of free pages into which the drum can read.

Because of slot conflicts, this rate can only be approached. A classical probability exercise known as the Urn-model Occupancy problem gives us the result.[3] With p slots and k requiests, the probability, P_i, that exactly i pages ($1 \leq i \leq \min(p,k)$) can be read in one revolution is given by

$$P_i = \frac{p!}{(p-i)!\, i!} \sum_{v=0}^{i} (-1)^v \left(\frac{i!}{(i-v)!\, v!}\right)\left(\frac{i-v}{p}\right)^k.$$

The average or expected number of pages which the drum can read, given k, is

$$M_k = \sum_{i=1}^{k} iP_i.$$

If M_k is less than $\frac{p}{1+f}$, there will be sufficiently many unused slots in each revolution to do the necessary page writes. Slot conflicts pose no concern during writing because the supervisor chooses the output location after the read requests have been scheduled. Thus, for queue of length k, the drum can deliver

$$Q_k = \min\left(M_k, \frac{p}{1+f}\right).$$

pages per revolution. The total effective swapping rate is the mean of the Q_k weighted over the probability distribution of k. (This distribution can be determined empirically or by making assumptions about the program structure.)

In Figure 3, we plot M_k as a function of k and f for several examples. The case $p = 9$ and $T = 34$ msec is the TSS drum system, where 9 pages are recorded on two tracks (the length of a page is 4096 bytes = 1024 words). The case $p = 4$, $T = 17$ msec, is the same drum reformated to waste some space in favor or reducing the length of the cycle. The final case is a hypothetical drum which can hold only one page per track, but rotates with $T = 3.4$ msec, the maximum speed of the channel.

We see from the figure that the hypothetical drum requires only one task queued for service in order to maintain the maximum swapping rate, but the others require 3-4 tasks and 8-10 tasks in the queue for $p = 4$ and $p = 9$, respectively. If k is smaller, then some slots are idle while tasks wait because of slot conflicts.

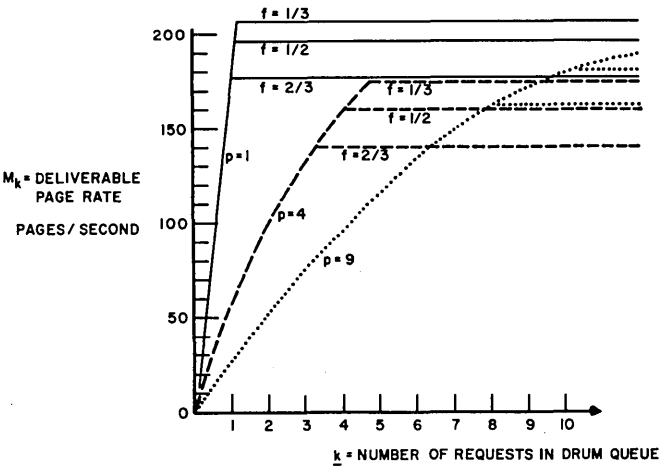


Figure 3 – Deliverable paging rate

But those slots must still pass the read-write head, and this wastes channel time. Our initial assumption was that channel time is at a premium and cannot be lost; so if we use one of these drums, we must operate with a large k. Main core must then be big enough to contain the many tasks which make up the drum queues plus some tasks which are ready to execute. We will see shortly just how much core this must be.

The effect of adding drums on separate channels can be seen from this model. Although they would not be synchronized, they could be coordinated and scheduled together. Then for n drums, the maximum page rate is $\frac{n \cdot p}{1+f}$. One could also plot a graph of M_k versus k, and one would find a more rapidly rising curve than in the case of a single drum. This can be attributed to the fact that there are n times as many slots passing the heads in the interval of time T, and thus less chance of a slot conflict for a given k.

Requirements of mean free paths

We would like to maintain complete processor-swapping overlap. To do so, the total amount of CP time necessary to do the work must be at least as great as the total amount of channel time necessary to support the CP. From this, we can determine the minimum mean free path of tasks. If k is the drum queue length, then in a revolution of time T, M_k tasks are delivered to the CP, on the average. Their total execute time must exceed T so that the mean free path must exceed T/M_k. In Figure 4, we plot this value as a function of k. For large k, there is little difference between the three cases; but for k, equal to two or three, the hypothetical drum restriction is less

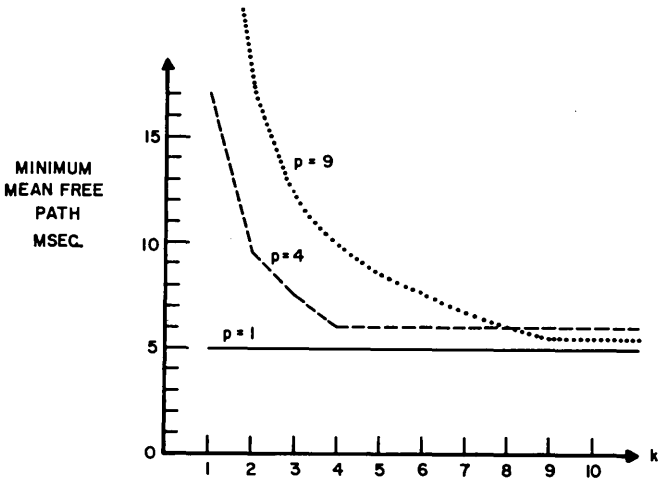


Figure 4 – Minimum allowable mean free path

by a factor of two to four. (We will see that LCS, when operated with a core-to-core channel is a realization of this hypothetical machine.)

From this calculation, we can discover how short our average time slice can be. An example will help. Suppose that tasks exhibit the behavior of Figure 1 with the knee of the curve at one millisecond and ten pages. Suppose that the drum can deliver 100 pages per second. Then the time slice must be at least 100 milliseconds, for the drum can "set up" no more than ten tasks per second. Any attempt to run with a smaller time slice will automatically generate CP idling.

It should be pointed out that this discussion deals with averages, and not the statistical fluctuations of random variables. Thus, our requirements are only necessary; they are definitely not sufficient. For example, if all tasks with free paths less than the mean were scheduled together, followed by all others, first the CP would idle, then the channel. On the other hand, if there is one task with an infinite free path, then there is no restriction on the others provided that one is not bumped from core. When a more comprehensive model and some empirical evidence are available, we will be able to generate sufficient conditions for complete overlap.

The lockout effect

Suppose we are able to keep the CP ahead of the channel. We will make an estimate on the amount of core necessary to support this—i.e., the amount of core necessary to contain the tasks which are in the page-wait state. To do this, we want to calculate first the amount of time a task spends waiting for a single page, a function of the rotational delay of the drum. Suppose for the moment there are no slot conflicts. Then the wait time is

$$w = t_1 + j \cdot \tau$$

where

t_1 = a uniformly distributed random variable representing the time to the beginning of the next channel program;

$$0 \leq t_1 \leq T.$$

j = the slot position of the page requested, uniformly distributed over

$$1 \leq j \leq p.$$

$\tau = \frac{T}{p}$ = the transmission time of a page.

Then the mean wait time is

$$\bar{w} = \bar{t}_1 + \bar{j} \cdot \bar{\tau} = \frac{T}{2} + \frac{P+1}{2}\bar{\tau} = T\left(1 + \frac{1}{2p}\right).$$

If n pages are requested by a task during a time slice, it spends n time the value $\bar{w}$ in the wait state. When we consider slot conflicts, this value could increase by as much as 50%. However, we must know the stationary distribution of k in order to make explicit calculations.

Let a task begin a time slice. It demands its first page and waits an expected time of $\bar{w}$ milliseconds, and *ties up one page of core for that time.* A short while later it demands its n^{th} page, it ties up n pages of core for $\bar{w}$ milliseconds. During the entire time slice, it causes a total of page-milliseconds of core to

$$1.\bar{w} + 2.\bar{w} + \ldots + n.\bar{w} = \bar{w}\sum_{i=1}^{n} i = \bar{w}\,\frac{n\,(n-1)}{2}$$

have been devoted to waiting for the drum to spin. These page-milliseconds are not available for any other purpose, but only to support the task for its slice of length, say, t. In other words, we must expend $\bar{w}\frac{n(n+1)}{2t}$ page-milliseconds of core in page waits to get a millisecond of useful work. I.e., in complete processing-swapping overlap conditions, *an average of* $\frac{\bar{w}n(n+1)}{2t}$ *core pages are tied up waiting for the drum at all times.*

Let us calculate an example using the SDC data of Figure 1 applied to the 360/67. From the figure, we see that if t = 160 milliseconds, then $n = 24$ pages. For a conservative estimate, we ignore slot conflicts and take $\bar{w} = T(1 + \frac{1}{2p})$. Then

$\bar{w}$ = 36 msec if T = 34 msec, $p = 9$;
$\bar{w}$ = 19.1 msec if T = 17 msec, $p = 4$; and
$\bar{w}$ = 5.1 msec if T = 3.4 msec, $p = 1$.

The total wait-time for all 24 requests is
860 msec if T = 34, $p = 9$;
460 msec if T = 17, $p = 4$; and
122 msec if T = 3.4, $p = 1$.
(I.e., the setup time for a 160 msec time slice is 860 msce in the case of the nine-slot drum. During this time, others are computing, but the conversational user will see the delay in the form of poor response time.) The locked-out core amounts to
62 pages for T = 34, $p = 9$;
33 pages for T = 17, $p = 4$; and
8.8 pages for T = 3.4, $p = 1$!
With regard to slot conflicts, observe that each conflict adds only 3.4 milliseconds to $\overline{w}$ in the case of $p = 1$, but it adds 34 milliseconds to $\overline{w}$ in the case $p = 9$. Thus, the high probability of conflict on the hypothetical drum is offset by the expense of conflict on the nine slot drum. In actual operation, we would find that the number of pages locked out would be greater than the values we calculated by roughly the same factor for each of the three cases.

These costs in core to support demand paging with conventional drums are very high, and nearly all of it can be attributed to the rotational delay. The analysis can be generalized and the same kind of results can be obtained for file operations from disks. It can also applied to non-360 time-sharing systems to obtain similar results. Neither must one restrict himself to fixed sized pages or a demand-paging concept. The principle is clear that the best device is one which rotates with a cycle of no more than the transmission time of the smallest swap, or one which does not rotate at all—i.e., LCS.

Affinity paging from a drum

Several proposals have been made to improve drum performance over the demand paging case. Most of these involve swapping groups of pages which have an affinity for each other. I.e., if one page is requested, the supervisor recognizes that certain others will be requested with a high probability and initiates the swaps for those at the same time. In this way, the wait time for several requests is overlapped, and the amount of core necessary to support the drum is cut by an appropriate factor. Among the proposals are that of swapping whole programs at once (CTSS at Project MAC), reading in at the beginning of a time slice all of the pages which the task used during the previous one, or maintaining a set of links in the page tables in the supervisor which relates every page to its companions.

One can either require that all related pages be written in contiguous slots on the drum or allow them to be located randomly; each has advantages. In the former case, there is no possibility of slot conflicts between pages of the same group, and there need be a delay of at most one rotation to access them all. However, if two tasks each request groups of pages, and the sequences of slots for these pages overlap, one task has a very long wait while the other reads plus an additional wait while the drum spins to the proper place. The latency for two independent requests cannot be overlapped as it could in the demand paging case. Furthermore, when the system must write pages, it must find a sequence of slots long enough to contain all of the group. Our analytical techniques are not sufficient for calculating the deliverable page rate or the amount of locked out core under this configuration. Neither can we determine here whether the advantages offset the disadvantages.

In the latter case, where pages are located randomly on the drum, the calculation of the deliverable page rate for the demand paging drum applies. If a task demands m pages, this is equivalent to m tasks each demanding a page, at least from the point of view of the drum. In fact, we increase the effective value of k in those calculations, increasing the swapping rate, without adding more tasks to core. The fact that the latency of the m pages is partly overlapped also reduces the total amount of core time necessary to support a time slice of computing. Clearly, this type of affinity paging is an improvement over demand paging.

If swapping is done from LCS, affinity paging is useful only if the swapping channel is not busy. Since there is no latency time, the amount of waiting for two pages is twice that of one page. If the channel is free, then we can initiate a swap for the task which is executing before it demands that page. But if other tasks are demanding immediate service, they must be handled first to reduce the chances of the CP idling for lack of work in the ready state.

We should point out that we assumed the system supervisor has some way of recognizing page affinities. This might result from a heuristic operating in a demand paging environment which "learns" which pages are related. We might require the user of his compiler to specifically define the relationships before run time. Or some other method could be used. In any case, affinity paging appears to be a necessity for systems committed to using drums for swapping. But the benefits of using LCS go beyond the realization of the hypothetical one-slot drum, as we will see.

Comment

We have seen that a drum cannot deliver pages to tasks at the maximum rate of the swapping channel

unless we allow large queues for service to build up. This is because of its inherent rotating nature, which wastes channel time with slot conflicts. Only in the degenerate case of $p = 1$ is it possible to get maximum performance with short queues. The rotational delay causes tasks to spend a disproportionate amount of time in the page wait state, relative to the computing they do; and this eats up core. Affinity paging reduces these problems, but so does LCS with a swapping channel. Carnegie has chosen the latter path.

The concept of giving each user a slice of CP time out of an operational cycle of say, one or several seconds, has been very popular in time-sharing circles—indeed, some will take it as the definition of time-sharing. But our analysis has shown that there are other costs in providing computer service, such as core space-time, and channel time. If the user demands these in unusual proportions, the system can get bogged down. Perhaps when we allocate his time slice, we should also allocate core time, channel time, I/O time, and other resources. If he exceeds any one allocation, or perhaps if he exceeds some function of them, his turn in the operational cycle should be considered ended. If, for example, we granted a core time slice instead of a CP time slice, he would be entitled to squeeze as much computing out as he could, provided he did only little paging. Or he would be entitled to swap heavily but only compute a little. But whatever the allocation scheme, it should be designed to keep the demands for the various scarce resources in proper balance to avoid waste.

We have examined the system costs with respect to channel time and core space, but we have not discussed the effect of fixed size pages. In the 360/67, this is irrelevant because the hardware is designed for 4096 byte pages. In the larger domain of time-sharing, variable size pages should be considered, for it is apparent that a lot of swapping is done to gain access to only a few words. If only those words were swapped, it might be possible to save significant amounts of overhead. LCS will provide us a crude approximation of this ability by allowing direct accesses to words stored there.

The Carnegie LCS-oriented system

In place of a drum, we provide an equal amount of bulk core storage with 8 μsec cycle time, addressed as an extension of main core. A core-to-core channel (known as the Storage Channel) allows information to be block-transferred from any memory location to any other, independent of and overlapped with CP operation. This channel behaves exactly like any other I/O channel on the system and operates slightly faster than a drum channel. Thus, we can treat the LCS as an on-line device which does the page swapping.

Because there are no rotational delays, the channel can access any page as soon as it finishes a previous operation. In this sense, it is like the one-slot hypothetical drum which we discussed above. With it, we can enjoy the maximum possible paging rate without maintaining large queues of idle tasks. The restriction on the mean free path of programs is less than it would be if we allowed only a small number of tasks in core and a low drum paging rate. Perhaps most significantly, we do not need to maintain the extra 30 to 60 pages of fast core to drive the drum. Each task spends less time in main core waiting for pages and, consequently, gets through its time-slice in less real time.

An equally important benefit of LCS is the fact that it is addressable directly by the CP as an extension of main memory. It is possible that when a task references a page, it may not be worth the overhead to swap it. For example, only a few words from the page may be needed for the immediate processing. If the page resided on a drum, it would have to be swapped, no matter what. But if it resides in LCS, then the page tables can be set so that the CP thinks it is in core and accesses the information directly, word by word. The cost involved is the CP degradation due to the long cycle time of bulk core; the savings come from the fact that there is no swap or other system overhead. The trade-off point is determined by the usage of the page.

For example, suppose that it takes 500 μsec of CP overhead to process a paging interrupt. Suppose also that it takes 4 μsec to fetch a double-word in LCS, over and above the normal access time in fast memory. Then by not swapping a page, we can buy 125 direct accesses to LCS; we leave the Storage Channel free for other use; and we do not tie up a page of fast core. One can easily think of many data structures which occupy a thousand words, but are not referenced more than 125 times per time slice: e.g., a data set catalog, a portion of a large IPL-V list, or a string of post-fix code to be executed by an interpreter.

A. L. Sherr has pointed out that there is much to be gained from small page sizes, provided the overhead of swapping can be counquered.[4] What the LCS configuration does is provide two different page sizes —the one is the 4096 byte page which is swapped, but which remains available throughout a time-slice; and the other is a double-word "page" which is accessed only at the cost of an LCS cycle, but which is highly transient and must be re-accessed each use. Each serves its own function; the larger supports normal computing, while the smaller provides effi-

cient access to large data structures. This duality decreases the rate of demand for swapped pages by the task, or equivalently, increases the ability of the system to meet the demand rate.

The question of how to decide which pages should be swapped and which should be accessed directly is an unsolved one, not unlike the question of defining affinities between pages. We propose no solution here, but we observe that a simple method is to require users and compilers to identify which pages are which. Certain stimuli can be created to induce users to do this efficiently, but caution is important. Our object is to fit the machine to the users, not the users to the machine.

The machine configuration

Figure 5 shows the single-processor simplex system initially installed. The machine differs from a

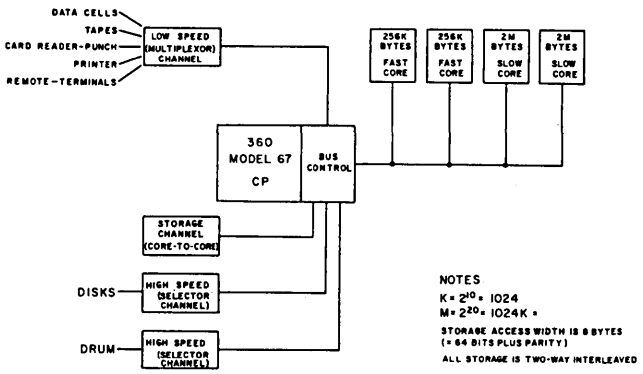


Figure 5—Simplex 360/67 with LCS

standard 512K byte 360/67[5] by including over four million bytes of IBM 2361 Large Capacity Storage and the Storage Channel for core-to-core transfers. No drum will operate in the system. The LCS has an 8 microsecond memory cycle, a 4 microsecond access time, an 8 byte (64 bit) data width, and is two-way leaved. Thus, it can support a data rate of two million bytes per second. The Storage Channel is controlled by the same type of channel program as other 360 channels, and can transmit 1.6 million bytes per second (400 pages/second) between locations in core.

This machine costs more than the equivalent drum machine. Suppose c is the cost of a basic 512K 360/67 with one drum and a normal complement peripheral gear. Then, the drum and its channel represents about $7\%c$, while the same capacity of LCS (with Storage Channel) costs 25% c. The amount of fast core locked out by the drum, but which is available for other use in the LCS system, cost 6%-12% c. The same basic system with LCS cost

$$c - (\text{drum}) - (\text{locked out core}) + \text{LCS}$$

or 7%-12% more than the drum machine. However, for this, we get a faster paging rate and a much more flexible machine.

A duplex system scheduled for operation at Carnegie in the future is shown in Figure 6. The system has two Central Processors and two Channel Controllers, each with an independent access path to memory modules. Attached to each Channel Controller is a Storage Channel as well as conventional multiplexor and selector channels. There are no no drums, but other I/O devices are configured in the same way as on a standard Model 67.

A half-duplex system is essentially half of the system in Figure 6; i.e., one CP and one Channel Controller each with an independent access to memory.

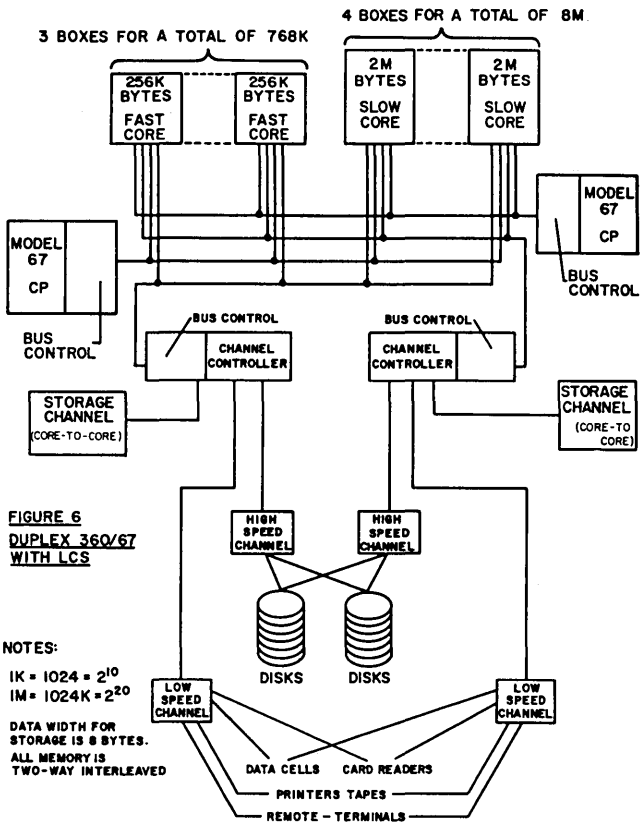


FIGURE 6
DUPLEX 360/67 WITH LCS

Figure 6—Duplex 360/67 with LCS

It is logically identical to the simplex system, but functionally the conflicts between I/O, Storage Channel, and CP are resolved at the core modules rather than at the storage bus. The resolution circuitry introduces a delay of 150 nano-seconds per memory access, which degrades CP performance from specifications. However, high I/O rates and Storage Channel transmission *also* degrade CP performance in a

simplex system by causing memory bus interference. The trade-off point between the simplex and half-duplex systems was determined by simulation[6] and corroborated by analysis of manufacturers' reports. Essentially, if the utilization factor of the Storage Channel is less than approximately 50%, then the simplex system gives better performance, Otherwise, the half-duplex system gives better performance. There is no suitable half-duplex machine available to compare to the simplex machine in order to verify these results. If the present analysis holds up, Carnegie will upgrade its initial system to a half-duplex machine when practical.

CONCLUSIONS

From the analysis presented in this paper, Carnegie Institute of Technology has discovered that it cannot live with time-sharing on a drum-oriented machine with demand paging. The swapping rates which it can support are too low for practical operation, unless we admit a lot of extra, expensive main core. Paging from bulk core, where there are no slot-conflicts, yields a factor of two to four better performance. At the same time, the LCS allows a reduction in the page demand rate because it can be directly referenced. We have also discovered that the inherent rotational delay of the drum has the effect of withdrawing 30 or more pages of memory from usable core at all times. Then in itself is expensive.

We recognize that, like all models, our model is only an approximation. However, we feel confident that we have effectively handled the first major problem of understanding what is required to make time-sharing work. Whether we have made forward progress or whether another equally big problem is hiding behind this one, only time will tell.

All of the improvements which come with LCS are needed, and they more than justify the extra expense. A rotating memory is too inflexible and inaccessible to support a comprehensive time-sharing facility, just as it was found to be too inflexible and inadequate as the main memory for second generation computers.

In fact, before the "fourth generation," we will probably find that large non-rotating memories assume increasing importance both as swapping devices and storage for large on-line files. Carnegie cannot afford to wait for this trend.

ACKNOWLEDGMENT

I wish to thank Professors Allen Newell, Alan Perlis, and Mr. David Nickerson for their valuable comments and their enthusiastic endorsement of the system. Mr. Michael Gold and Mr. Manuel Langtry were very helpful in clarifying the ideas and presentation. Miss Polly Breza, Mrs. Cynthia Yang, and Mr. Albin Vareha, who are responsible for the software modifications to TSS, have kept the discussion down to earth. Finally, Miss Joyce Nissenson has helped with the manuscript.

REFERENCES

1 J COHEN
A use of fast and slow memories in list-processing languages
CACM Vol 10 1967 p 82

2 G FINE C JACKSON and P McISSAC
Dynamic program behavior under paging
Proceedings—Nat onal ACM meeting 1966 p 223

3 W FELLER
Introduction to probability theory and its applications
Vol I pp 91 ff

4 A L SHERR
Analysis of storage performance and dynamic relocation techniques
IBM document TR-00-1494

5 C T GIBSON
Time-sharing with the IBM 360/67
FJCC 1966 p 61

6 K GRAHAM
Unpublished results Carnegie Institute of Technology 1966

Modular computer design with picoprogrammed control

by J. G. VALASSIS
Automatic Electric Applied Research Laboratory
Northlake, Illinois

INTRODUCTION

The subject computer is a 16-bit, one microsecond, integrated circuit computer implemented with picoprogrammed[1] internal control. (Picoprogramming is conceptually comparable to microprogramming but without the need for decoding logic.) The design provides for modular CPU expansion both in word-size and instruction-repertoire. In addition, it lends itself very admirably to post-design tailorability, which is the most important unique feature of its picoprogrammed control design.

Picoprogramming is realized by use of the MRYi-Aperture (MYRA[2]) element, a multiaperture ferrite device which is the basic building block of the instruction module. Each instruction module is a complete entity and is fabricated on a conventional printed wiring card that can be inserted in a conventional PC connector. Incorporation of a new instruction in the computer or alteration of an existing one is accomplished by the addition or substitution of the appropriate instruction module card.

Decoding of the contents of the machine instruction word provides the address for the selection of a particular instruction module. A selected instruction module generates the necessary serial control levels and pulses. The timing of these pulses has been charted to effect the appropriate data transfers within the machine's CPU to hardware-implement the micro-operations specified by a particular instruction (e.g., ADD, Y). Each instruction module can be thought of as the decoded output of a fast microprogrammed memory whose microinstruction-word contents dictate specific operations within the CPU to hardware-implement the specified machine instruction. The number of words of this virtual microprogrammed memory is dictated by the complexity of the particular instruction. For the ADD instruction, for instance, the equivalent of eight such microinstructions are necessary; and since two main-memory references must be made (one to fetch the next machine instruction and the other to fetch the operand), the maximum cycle time of the virtual microprogrammed memory must be 250 nanoseconds. In the MYRA picoprogramming element, this 250 nanosecond speed of operation is realized by generating sequential control pulses with pulse-widths of 250 nanoseconds. Hence, the total access time of the MYRA element must be 2 microseconds.

This paper will not concern itself with the computer architecture, which is comparable to contemporary computer organizations, but rather with the design aspects of its picoprogrammed control unit. The discussions within this paper will also point out design shortcomings of the MYRA concept and how novel techniques are employed to ameliorate inherent speed limitations of the picoprogramming MYRA element.

Computer control requirements

The computer has the general salient features shown in block diagram form in Figure 1. The hardware implementation of the instructions, specified by the program residing in the ferrite core memory, is accomplished by the internal logic control provided by the MYRA picoprogrammed control unit. Transfer control pulses within the CPU or the Direct Memory Access (DMA) unit with core and I/O are produced by specially flow-charted instruction modules.

In synchronous wired-control logic computers, these sequential transfer-control pulses are generated by the wired logic at a rate specified by the repetition frequency of a master clock. In microprogrammed logic, these sequential control pulses are generated by successive microinstructions contained in consecutive words in an internal fast control memory. Appropriate decoding of the contents of these sequential words produces the necessary control pulses to implement the operations specified by the microinstructions. Since picoprogramming is a step beyond microprogramming, in that the generated sequential control pulses are already decoded as they are produced by the MYRA picoprogramming module,

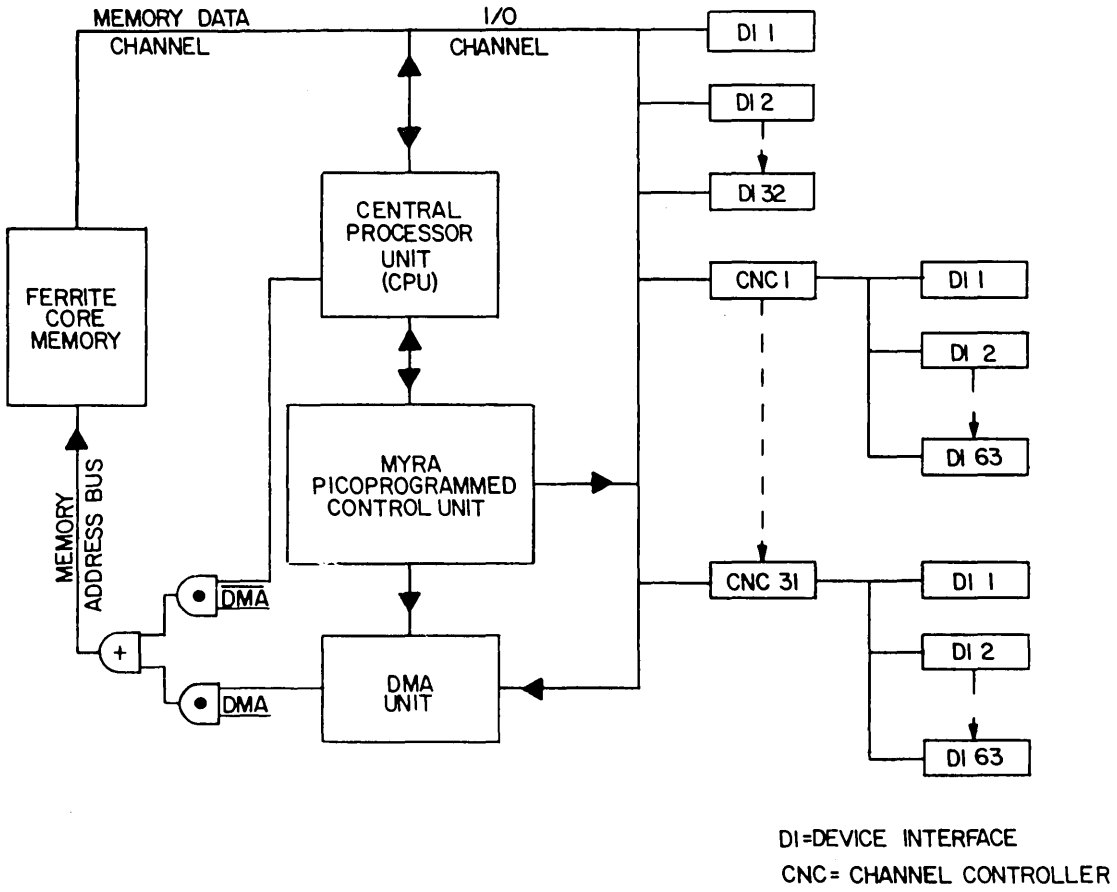


Figure 1 – Picoprogrammed computer configuration

charting of machine instructions in terms of microprogrammed flow charts is quite useful to formulate the control pulse-timing requirements to implement the respective machine instructions.

To demonstrate the direct correspondence of microprogrammed control to picoprogrammed control, the ADD instruction's microinstruction flow-charting is chosen as an example. Figure 2 indicates the necessary operations that must be performed in a sequential fashion to implement this instruction. The control levels required to effect execution of the specified microinstructions are shown in Figure 3. They are associated with Place Levels (PL), Accept Levels (AL) and Transfer Levels (TL) of registers and circuits in the CPU, shown in Figure 4.

There are two types of control levels generated by the instruction modules: unbuffered levels (e.g., BTL), and derived levels suffixed with the letter "D" (e.g., UPLD). An unbuffered control level is wired directly from the output of the instruction modules to the corresponding control point of the Central Processor. A derived control level is produced by a logic decision generator circuit. For example, the logic expression of the output of the UR Place Level (UPL) logic generator is:

$$(\text{UPL}) + [(\text{COND1}) \cdot (\text{ADRD})] + [(\text{COND2}) \cdot (\text{UPLD})]$$

This derived UPL control level is produced unconditionally with UPL, conditionally with derived control level ADRD as specified by the logic requirements of the 2nd microinstruction (COND1), or conditionally with derived control level UPLD as specified by the logic requirements of the 4th microinstruction (COND2). Since the two derived MYRA signals, ADRD and UPLD, are mutually exclusive, false generation of the derived UPL level is avoided.

Picoprogramming implementation of microinstructions

Selection of the appropriate instruction module is accomplished by decoding the contents of the instruction register (IR). The select-set condition of the MYRA element is an AND function of the OP-code,

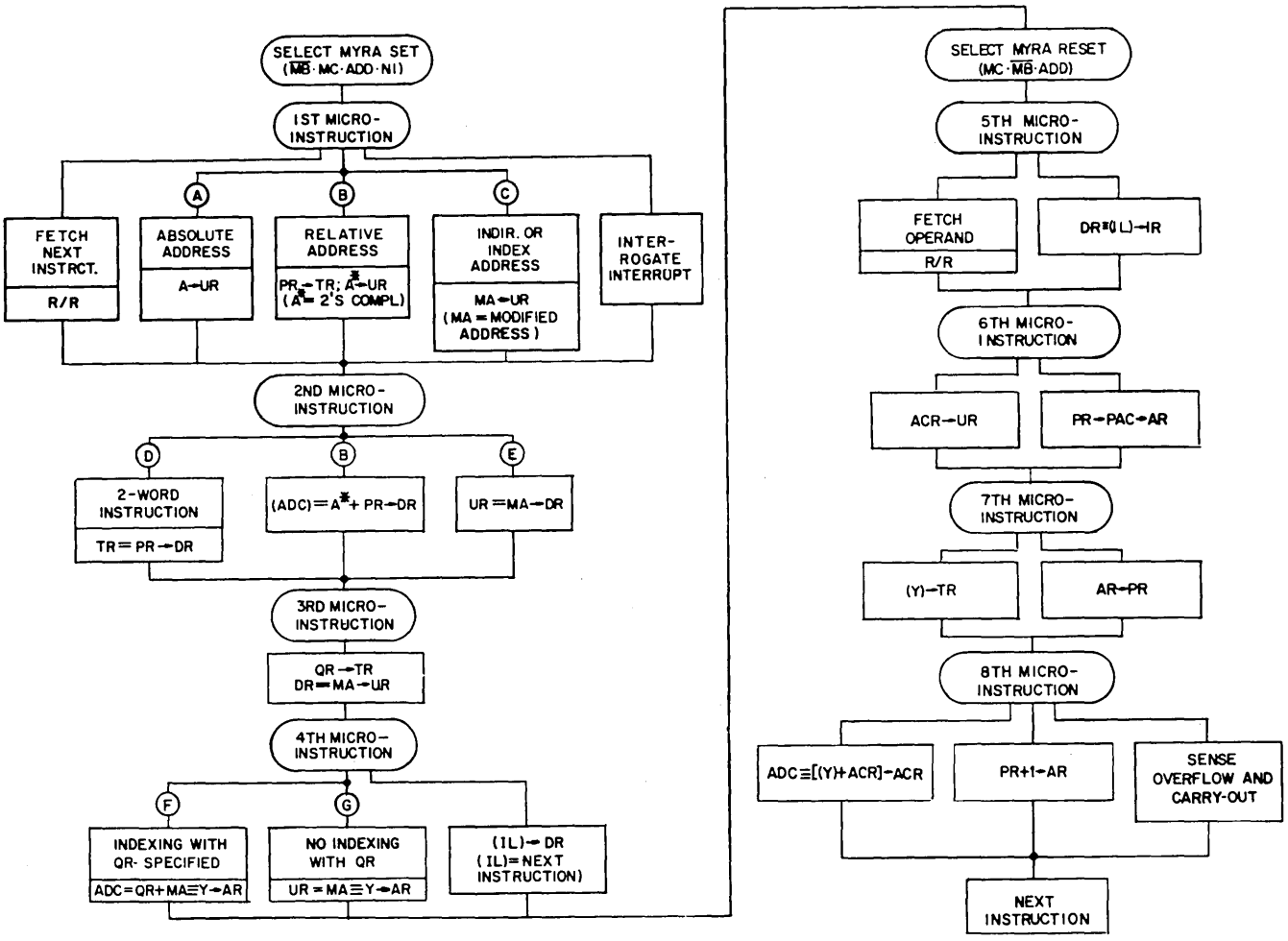


Figure 2—Microinstruction flow-charting of the ADD instruction

the memory not busy ($\overline{MB}$) signal and the MYRA control (MC) status. For the ADD instruction:

SELECT MYRA SET = (MC)·($\overline{MB}$)·(ADD) (NI)

where NI indicates that indirect addressing or indexing with respect to memory locations 0, or 1 is not specified. If such address modification is specified, it is implemented with picoprogramming control produced by the indirect addressing and indexing modules respectively before selecting the ADD instruction module.

When the Select-MYRA-Set condition becomes true, the MYRA driver of the ADD instruction module is turned on, and with the aid of the sustain set (SS) winding (Figure 5) switches the first 4 areas of the MYRA disk. Each area looped with a control winding will induce in the winding a 250 nanosecond pulse. Winding arrangements corresponding to control-level requirements to perform the first four microinstructions associated with the implementation of the ADD instruction are obtained directly from the pulse timing diagrams of Figure 3. Thus, the UTL control winding loops only the third area, while the BTL control winding loops areas 2, 3 and 4 of the MYRA disk.

The first four control levels of Figure 3 are necessary for the proper operation of the MYRA control circuitry and are common to all MYRA modules. All of the control pulses produced during the setting of the MYRA element are common to most of the memory-referenced machine instructions. This is evident from the fact that the first four microinstructions are performing address modification to evaluate the effective operand address (Y), while waiting for the memory to fetch the next machine instruction.

The last four microinstructions necessary to implement the ADD instruction are executed under control of serial control logic generated by the ADD instruction module during resetting of its MYRA element. Resetting the MYRA disk is initiated when the reset condition becomes true:

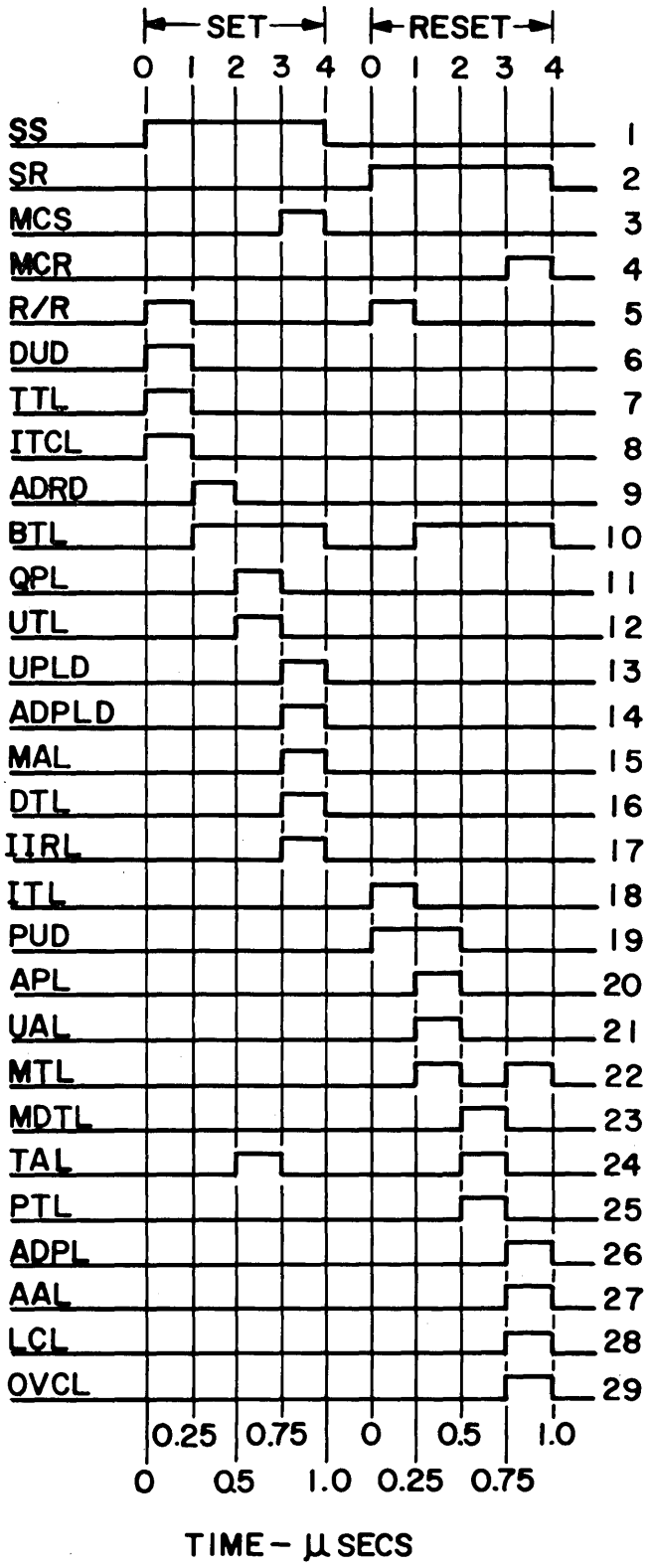


Figure 3 – Picoprogrammed control logic to implement the ADD instruction's microinstructions

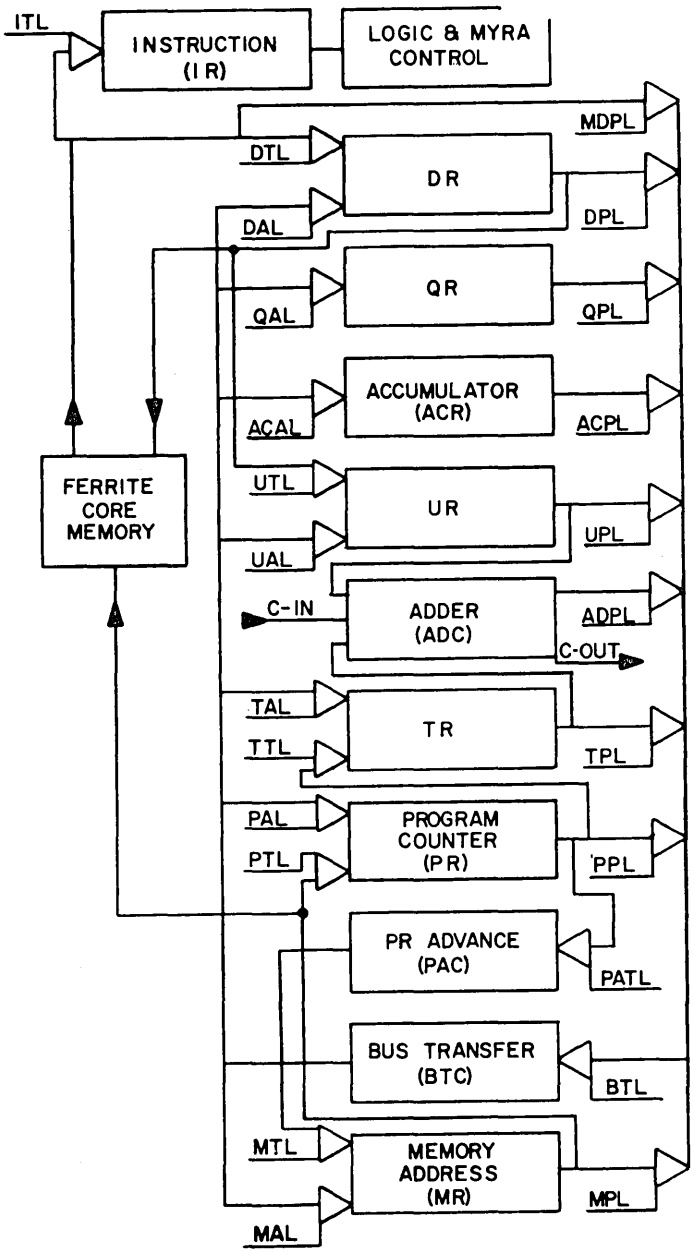


Figure 4 – Partial CPU configuration

SELECT MYRA RESET = (MC)·($\overline{\text{MB}}$)·(ADD)

The sustain reset (SR) winding allows switching of the disk up to the fourth area, thus returning the disk to its unselected state.

During resetting of the disk, the appropriate winding distribution produces the desirable control levels as dictated by the chart of Figure 3. The majority of these pulses are common to most of the memory-referenced instructions, since they represent housekeeping operations. These operations include advancing the program counter (PR) and loading the instruction register (IR) with the next instruction. Control levels 10, 20, 21, 26, 27, 28 and 29 of Figure 3 are the only unique pulses associated with the ADD instruction. The beneficial aspects of the small ratio of unique windings to common windings, approximately 1/10, result from the fact that mass prefabrication of classes of instruction modules is possible. To these prefabricated modules, the small portion of unique windings needed to fabricate the required instruction module can be added with a comparatively small time and effort. Thus, modular expandability of the machine's instruction-repertoire by the addition of a new instruction module, or modular alterability by substitution of an existing instruction module with a new one is readily realizable.

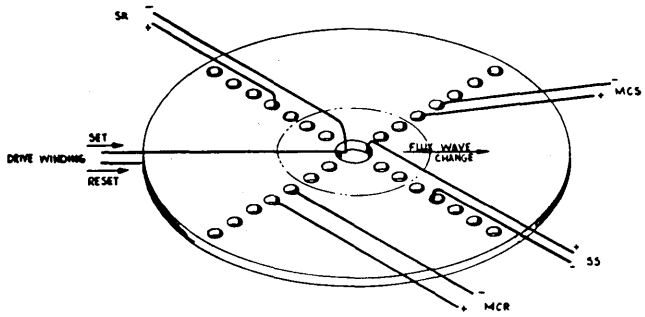


Figure 5 – MYRA picoprogramming element arrangement

Instruction module implementation

The switching operation of the instruction module is as follows. As soon as the "Select-MYRA-Set" level becomes true, the driver of the selected instruction disk is turned on. With the output transistor-switch in saturation, a step function voltage is applied across the MYRA drive winding causing a current ramp to flow through it. Thus, a propagating flux-change wave will result that will traverse the disk in an outward radial direction with a uniform velocity. As this wave traverses consecutive areas that have been looped selectively with picoprogramming control windings, the induced emf will produce the necessary serial picoprogramming control logic. The switching parameters of the disk have been chosen such that the produced MYRA winding pulses have voltage amplitudes compatible with the Sylvania SUHL I.C. logic which is used to implement the computer. This voltage level compatibility is significant because the output of the MYRA winding, due to its very low internal impedance (~10Ω), can be thought of as a virtual high fanout logic gate with a fanout factor in excess of 50. High fanout is a requirement in register control levels where all 16 bits of registers in the arithmetic unit are to be loaded with new information.

When the flux-change wave has traversed the fourth area, the zero state of the SS level causes the set driver to turn off. Thus, the current ramp that flows through the set winding drops rapidly to zero. The MYRA control set (MCS) winding that loops the fourth area of the MYRA disk has conditioned the MYRA selection unit to receive the succeeding reset signal to switch (reset) the selected MYRA disk to its original state.

Logic control levels are produced both during setting and resetting of the disk as dictated by the timing chart to perform the microinstruction operations. An ORing gate arrangement is used to chain MYRA windings associated with a particular control signal. This ORing arrangement, a 2-input diode OR-gate per MYRA control level, makes possible the utilization of control pulses that are produced during either setting or resetting of the disk.

The switching constant (S_W) of the ferrite MYRA disk material will yield a good resolution at speeds below 500 nanoseconds per area. In this serial logic arrangement, resolution is referred to as the ("one"/"zero") ratio in the worst-case condition where a logical "one" is both preceded and followed by logical "zeros." At the speed requirements of this computer (250 nanoseconds per area), the resulting resolution is so poor that the produced ("one"/"zero") ratio of 2 cannot be used for picoprogramming control purposes. The degradation in resolution at high switching speeds results from flux-change-wave broadening that causes several adjacent areas of ferrite material in the radial direction to switch simultaneously. As the rate of the magnetic-field build-up increases with an increase in current ramp slope, impulse switching conditions prevail and the ferrite material in the eighth area of the disk starts switching, while ferrite material in the first area has not yet been completely switched to its reset state. It is obvious then that such flux-change-wave broadening can be prevented if an inhibiting arrangement of areas beyond the one associated with the flux-change-wave front is devised.

The biasing winding arrangement of Figure 6 accomplishes the desirable inhibiting function. Under the assumption that the internal impedance of the current driver is small, the rate of change of flux with time is constant. Therefore, the current through the biasing winding for a given area (radial depth) is a function of the biasing resistor (R_B), the amplitude of the step driving voltage (V_D), and the number of turns (N_D) of the drive winding. As a function of the radial position of the flux-change-wave, in terms of areas in the radial direction (n), the biasing current is proportional to n or

$$I_B \cong \frac{n}{R_B}\frac{V_D}{N_D}\,;\ n = 1,2,\ldots 8.$$

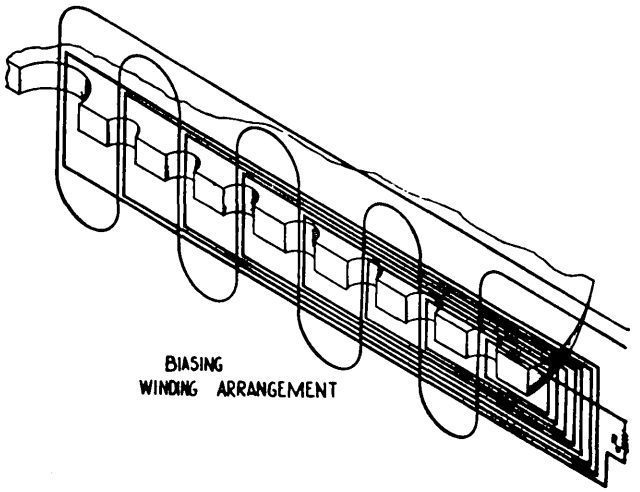


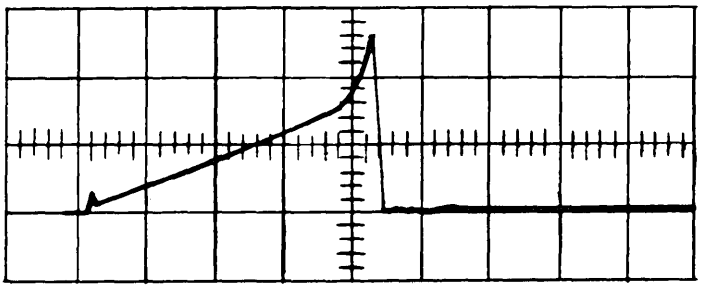
Figure 6—Cut-view of a biased MYRA-disk radius

Since the driving current ramp has a similar linear variation with n, knowledge of the switching parameters of the ferrite material (S_W and H_c) will allow determination of the value of R_B for a given V_D and N_D to produce optimum serial resolution at the specified speed of operation of the MYRA element.

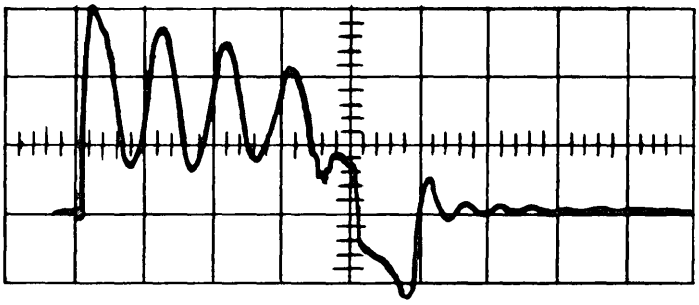
The effect of the biasing winding on the resolution of the MYRA serial control logic is depicted by the oscillograms of Figure 7. The picoprogramming code 10101010 was chosen as the worst-case resolution condition. From the oscillograms it can be seen that an effective discrimination range improvement in resolution of about twenty to one is obtained.

The outputs of the three picoprogrammed windings to effect the micro-operation (QR → TR) specified by the 3rd microinstruction of the ADD instruction are shown on the oscillograms of Figure 8. The fourth pulse which is necessary to reset TR and gate the new information into TR is not directly derived by MYRA windings but is logically produced from the leading edge of the TAL pulse.

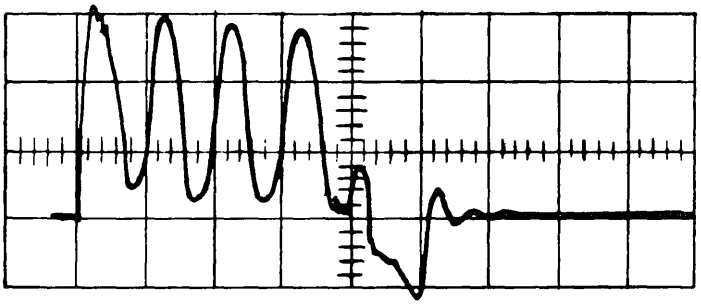
Instruction modules are packaged as single units on conventional double-sided PC cards. Two types of such cards are employed in the control unit of the computer: the single-disk instruction module (Figure 9) which contains the MYRA disk and two drivers (a set driver and a reset driver), and the double-disk instruction module (Figure 10) which contains two MYRA disks and three drivers (two separate set drivers and a common reset driver). From economic considerations, double-disk modules are preferable to the single-disk, because a saving of one driver and one PC card per two instructions is realized. From the



(a) DRIVE WINDING
CURRENT RAMP
13 AMP. TURNS / DIV.

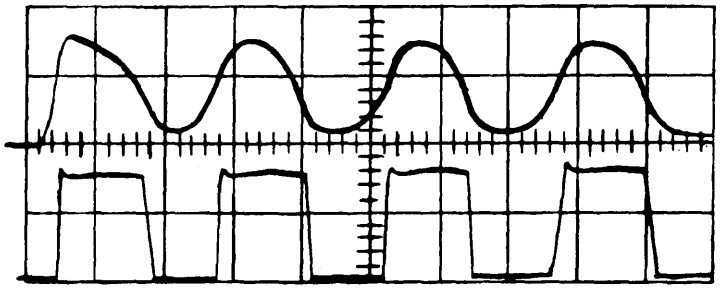


(b) "10101010"-PICOPROGRAMMING
MYRA WINDING
(UNBIASED)
1 V / DIV.



(c) "10101010"-PICOPROGRAMMING
MYRA WINDING
(BIASED)
1 V / DIV.

(a),(b),(c) ; 0.5 MICRO SEC. /DIV.



(d) "10101010" MYRA WINDING

(e) "10101010" SUHL I.C. OUTPUT
2 V / DIV.

(d) ; 0.2 MICRO SEC. / DIV.

Figure 7—Effect of biasing winding on serial pulse resolution

modularity and diagnostic point of view, however, single-disk cards that are associated with single instructions only are preferable. This is obvious from the fact that an inoperative instruction can be immediately traced to the respective module and diagnosed instantly, with no regard to ambiguities introduced from possible interaction of the adjacent instruction sharing the same module. In the computer, an equal number of single- and double-disk modules were used

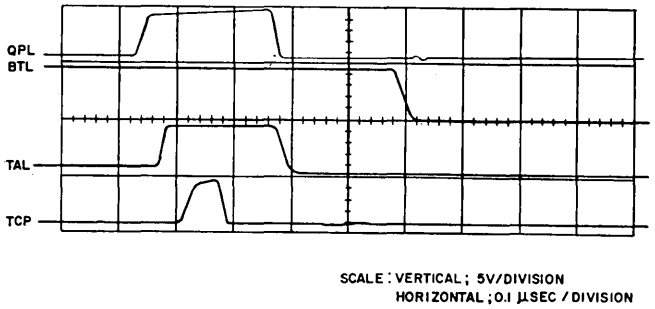


Figure 8—Sample microinstruction picoprogrammed control requirements

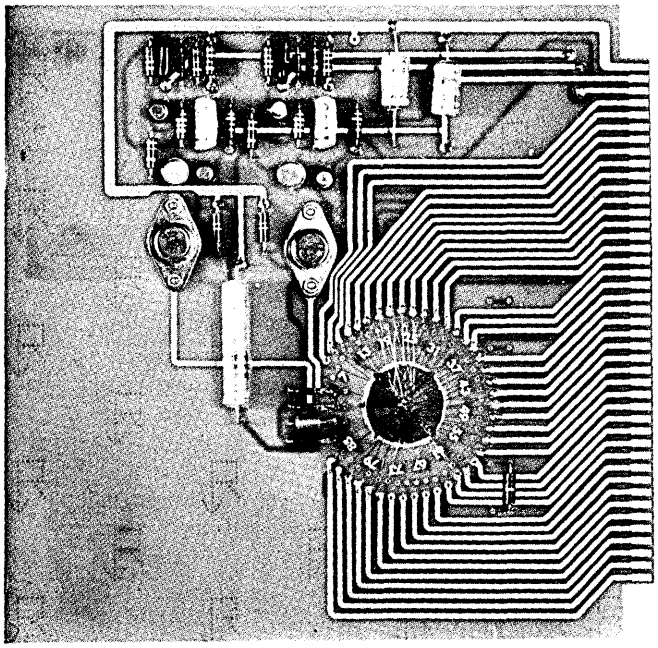


Figure 9—Single MYRA disk instruction module

Design features of picoprogramming

The modular expandability in terms of instruction repertoire of the picoprogrammed computer has already been discussed. The post-design tailorability of the machine by introducing or modifying features will now be considered.

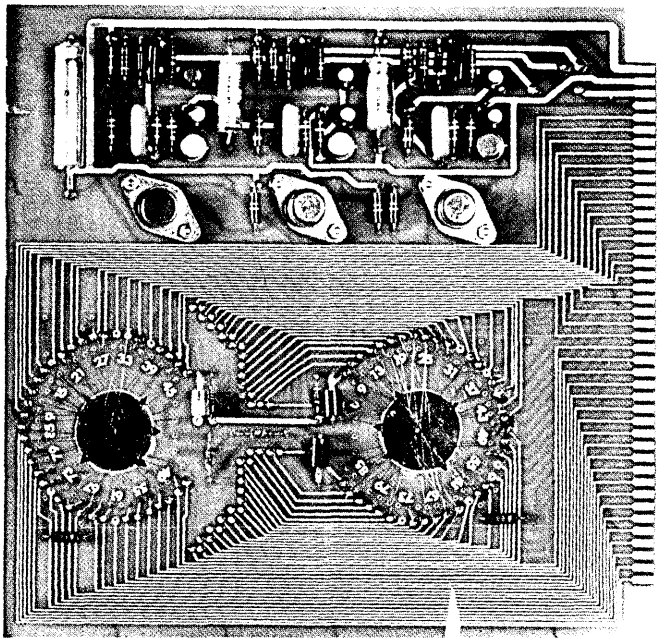


Figure 10—Double MYRA disk instruction module

It has been indicated that the computer has three index registers. One is a hardware register (QR) residing in the arithmetic section and the other two are the 1st and 2nd word locations of the ferrite core memory. When indexing with respect to a core index register is specified by the machine instruction, a special indexing MYRA module is employed. This module is accessed before the main instruction module is selected and address modification with the specified core index register is performed. This type of indexing is time-consuming in that it requires 1.4 microseconds more time than indexing with the QR which is in the arithmetic unit. With the present design, a future replacement of the core index registers with hardware is possible by the incorporation of an I.C. card that contains two 16-bit registers in the Central Processor and the elimination of the indexing MYRA module.

The incorporation in the computer of the optional Direct Memory Access (DMA) feature is also done by the addition of the DMA control unit and an I.C. card per required controller. This DMA control unit consists of three MYRA modules—one to initialize the selected DMA controller (load the block length, and the starting address) and one each to transfer data between the I/O device and the memory.

Cyclic instructions, like the hardware multiply and divide, are incorporated by the addition of four MYRA modules. The first initializes the instruction. The second partially implements the instruction by

performing seven successive test-add-shift operations. The third completes the implementation of the instruction by performing the last eight successive test-add-shift operations required to multiply two 16-bit numbers. The fourth finalizes the instruction.

CONCLUSIONS

The MYRA picoprogramming control technique was applied successfully in the design of a full-scale real-time control computer. Instruction modularity and post-design tailorability afforded the design by the picoprogramming concept were demonstrated. The former provides instruction-repertoire expansion or alteration in modular form, while the latter facilitates modular incorporation of machine features.

Operation of the picoprogramming MYRA element at speeds beyond its inherent acceptable resolution range was made possible with the incorporation of the unique biasing winding technique. This technique makes possible the application of the MYRA element in other serial logic designs as well.

The speed of operation (1 μsec. cycle-time) and the complexity of the computer (15 memory-referenced instructions, and 95 augmented OP-code non-memory-referenced instructions) made it an excellent worst-case-design vehicle. At the same time, it demonstrated that full advantage of the inherent useful properties of the MYRA picoprogramming concept (high fan-out, instruction modularity, diagnosability, etc.) is achieved with less complex CPU designs. In particular, the technique could prove very beneficial in multiprocessor design approaches where each satellite processor is relatively simple and is requested to handle only specialized types of data processing as part of a very powerful master processing unit that assigns specialized processing tasks to these satellite picoprogrammed processors.

ACKNOWLEDGMENTS

The author wishes to thank E. L. Scheuerman and J. E. Fulenwider for their helpful suggestions during the preparation of this manuscript. The efforts of J. R. Holden, who conceived and developed the biasing winding technique, are especially acknowledged. The ferrite processing contributions of Dr. M. E. Dempsey's materials group, and W. A. Reimer's efforts in packaging the MYRA control modules are greatly appreciated.

REFERENCES

1 B E BRILEY
Picoprogramming: a new approach to internal computer control
Fall 1965 Joint Computer Conference Vol 27 Part I Spartan Books Inc Washington D C 1965

2 B E BRILEY
MYRA: A new memory element and system
Proc 1965 Intermag Conference

Intercommunication of processors and memory*

by MEL PIRTLE
University of California
Berkeley, California

INTRODUCTION

Many computer systems include one or more high transfer rate secondary storage devices in addition to numerous input-output (I/O) devices. When the processors which manage these devices (frequently referred to as I/O controllers or channels), together with the central processing unit (CPU), communicate almost exclusively with a single primary memory, as in the configuration illustrated in Figure 1, the problem of providing these processors with adequate data transfer capability becomes formidable.[1] Ideally, each processor should be able to transfer a datum to or from primary memory at its convenience without regard to the ability of the memory to accept or supply the datum at that particular moment, or the ability of the processor-to-memory transfer path (memory bus) to effect the transfer. Unfortunately, economic and technical considerations dictate that memory systems of the capability implied must be relegated to the role of standards with which more practical systems may be compared. With practical memory systems, the rate at which data can be transferred between processors and primary memory is limited by the transfer capabilities, or bandwidths, of the memory itself and of the memory busses over which the transfers are made. Furthermore, since the memory system is shared by several processors, care must be taken to keep performance from being degraded excessively by interference caused by simultaneous attempts on the part of several processors to utilize a facility, such as a memory bus, which is capable of handling only a single data transfer at any given moment.

To provide the required memory bandwidth, many memory systems currently in use are partitioned into several modules which can operate concurrently.[2] In addition, some systems employ memories having a physical word length which is a multiple of the logical, or CPU, word length.[3] To make this memory bandwidth available to the processors, these systems frequently employ multiple memory busses, as shown in Figure 1.[4] To make a memory reference in one of these multi-bus configurations, a processor first makes a request for the use of its memory bus; and, when this request is granted, uses this bus to transmit its memory reference request to the appropriate memory module. If either of these two requests are rejected, the processor repeats the procedure beginning with the request for the memory bus. To resolve conflicting requests for either a memory bus or a module, some type of priority mechanism is employed. Priorities typically are assigned to processors and to memory busses;[5] the former are used to resolve conflicting requests for a memory bus and the latter to resolve conflicting requests for a memory module.

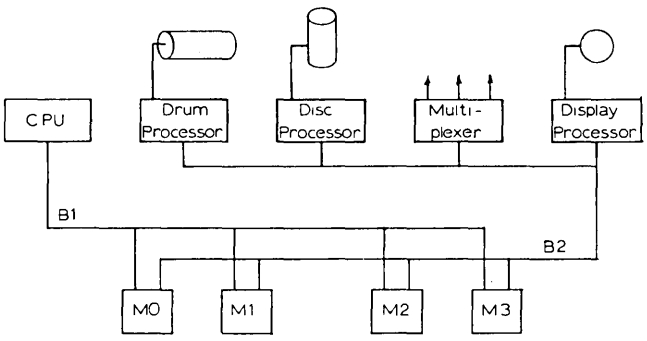


Figure 1 – General processor-memory configuration

Although these memory systems are generally able to satisfy the needs of the computer systems in which they exist, primary memory systems of even greater capabilities are required to meet the demands of newer computer systems which have faster CPU's, secondary storage devices with higher transfer rates, and a

*The work reported in this paper was supported by the Advanced Research Projects Agency, Department of Defense, under Contract SD-185.

high degree of concurrency of processor operation.[6] It is the purpose of this paper to discuss certain aspects of the problem of effective processor-memory intercommunication. A priority mechanism is suggested in which priorities are assigned to *requests*, on the basis of immediate requirements and conditions, rather than to *processors*. In addition, some memory system configurations and I/O processor buffering schemes are proposed which, together with the priority assignment mechanism, provide for high CPU performance and memory transfer utilization in an environment of simultaneous computation and high transfer rate I/O activity. Results of simulations of several configurations and priority assignment algorithms are presented in support of these suggestions.

General

To facilitate the discussions to follow, let us consider in detail the specific processor-memory configuration shown in Figure 1. This configuration includes a primary memory which is partitioned into four identical, independent modules having physical word lengths equal to the logical word length of the CPU. It also includes two memory busses labeled B1 and B2, the CPU having exclusive use of B1 and the several I/O processors sharing the I/O bus, B2. For these discussions, it is postulated that the transfer of a single word between a processor and memory requires the exclusive use of one memory bus and one memory module for a complete memory cycle (the time required to read and regenerate, or store, a word), and that the memory cycles are contiguous. Thus the memory system is synchronized: it accepts requests for the I/O bus and requests for memory references at regular intervals.

This memory system is capable of supporting simultaneous references to two of the four modules, one over each of the two busses. Thus, for example, the CPU can make a sequence of references to module M0 using bus B1 at the same time that the drum processor is making a sequence of references to module M1 using bus B2. Furthermore, if both of these sequences of references are made at a rate of 1/T, where T is the time required for one memory cycle, then 100% of the bus bandwidth available to each of the processors is utilized along with 50% of the memory bandwidth. However, simultaneous attempts to reference any one of the modules will result in the acceptance of only one of the requests, and simultaneous attempts by I/O processors to use the single I/O bus will result in the acceptance of only one of these requests. Since requests are sampled only once each memory cycle, a processor having a request rejected must wait for one memory cycle before having the request reconsidered; furthermore, the memory bus used for the rejected request cannot be used for any other request during the one memory cycle period.

The effective resolution of these potential request conflicts will be given considerable attention below, but before considering this problem in detail, let us question the general capabilities of the system under discussion. The system includes a CPU, a drum processor, a disc processor, a cathode ray tube (CRT) display processor, and a multiplexing processor which manages low demand I/O devices. Frequently, several of these processors will be active simultaneously. Some pertinent questions then are:

1. Does the memory have sufficient bandwidth to accommodate the transfers requested by the several processors?
2. Does each of the busses have sufficient bandwidth to accommodate the transfers requested by the processors attached to it?
3. Given that the answers to questions 1 and 2 are in the affirmative, can these bandwidths be effectively utilized?

The first two questions are purely theoretical, and easily answered: the first by comparing the memory bandwidth n/T, where n is the number of memory modules, with the memory request rate; and the second by a similar comparison between memory bus bandwidths and the memory request rates of attached processors. The third question is not answered so easily, however, since it involves not average request rates, but instantaneous request rates and the distribution of these requests over the busses and modules. For example, consider the configuration shown in Figure 1 and a situation in which the CPU and the drum processor are both active, the CPU making requests at a rate of 1/T and the drum processor at one-half this rate. Now, if the CPU and drum processor are referencing exclusively M0 and M1 respectively, there are no request conflicts and the required portion of the available bandwidths are used effectively. Also, if the CPU is referencing M1 instead of M0 on occasion, there is still no serious difficulty if the CPU refrains from making two consecutive requests to M1, since the drum processor can conveniently defer a reference for one memory cycle; although in this case, additional B2 bandwidth is consumed by rejected requests made by the drum processor. However, if both the CPU and the drum processor attempt to reference M0 exclusively, then either the drum processor is obliged to forgo the attempted data transfer or the CPU is forced to operate at one-half capacity. In this latter case only one-fourth of the memory bandwidth can be used although three-eighths of the memory bandwidth and

three-fourths of the total memory bus bandwidth are requested.

The question of memory bus utilization is similar to memory utilization except that in this case the only parameter to be considered is time since there is no opportunity for parallel activity on a given bus. The memory bus bandwidth can be easily utilized if each processor handling a synchronous device, such as a drum, is designed so that whenever it is transferring at a rate of R words per second it can make each word transfer at any point within the 1/R second interval between transfers. However, if each transfer must be made within some small portion of the 1/R second interval, as is frequently the case, then it is much more difficult to make full use of the bandwidth. To see that this is the case, consider a configuration in which three processors sharing a single memory bus must each make a transfer within a two memory cycle portion of their respective transfer intervals (the interval between transfers). Since each of these processors can defer a reference for at most one memory cycle, they cannot operate simultaneously, independent of their (non-zero) transfer rates, unless the operation of at least one of these processors is appropriately synchronized relative to one of the others. For only with synchronization can it be insured that the processors will not require three transfers in two memory cycles.

It is apparent that high utilization of memory and bus bandwidths are facilitated by transfers which are evenly distributed over time and over the several memory modules. The ease with which a given processor can provide an even distribution of memory references depends strongly on the manner in which the addresses are distributed among the memory modules and on the function performed by the processor. If the addresses of a memory consisting of n modules of m words each are distributed so that addresses 0 through m-1 are in module M(0) and addresses m through 2m-1 in M(1) etc., then a relatively even distribution of references over the modules can be effected by assigning to each processor a portion of the modules, and hence a memory bandwidth, commensurate with its transfer rate. However, this address assignment has the disadvantages that

1. The size of the memory assigned to a processor is directly proportional to its transfer rate, even though this relationship is frequently not consistent with other considerations.
2. In cases in which other considerations dictate the assignment of a single module to several high reference rate processors, the performance of one or more of these processors will be severely degraded, and
3. the memory bandwidth available to a sequential processor (a processor which transfers to sequential memory addresses) is limited to that of a single module.

These disadvantages are replaced by others of much less severity when the addresses are distributed over an n module memory so that address 0 is in M(0), address 1 is in M(1) and, in general, address X is in M(i) where $i \equiv X$ modulo n. As will be illustrated in subsequent sections of this paper, this address distribution together with appropriate memory bus configurations allows a given processor to be assigned any desired portion of the total memory bandwidth without regard to the size or location of the portion of memory assigned. This distribution, commonly referred to as interleaved addresses, does have two disadvantages: first, the operational failure of any of the modules nearly always renders the entire memory inoperative; and second, the simultaneous operation of several processors will almost always cause some degradation of processor performance. The former disadvantage is mitigated by employing reliable memory systems,* and results presented in subsequent sections show that the magnitude of the latter disadvantage can frequently be reduced to an acceptable level. All memories considered in the remainder of this paper will have interleaved addresses.

The function performed by a particular processor also influences the economic feasibility of distributing its references over time and over the memory modules. For example, a drum processor usually references sequential addresses at regular intervals, thereby distributing its references evenly over the memory modules and over time. A CPU, however, may frequently make several successive references to a single module or to a small number of modules, and it makes memory references at frequent, irregular intervals. More generally, there are four factors to be considered. They are:

1. The frequency of memory references
2. The ability of the processor to anticipate the need for a memory reference
3. The cost of deferring a reference
4. The natural distribution of references over memory modules

The importance of these factors will become evident in the later sections. At this point, it is sufficient to point out that processors making low speed, synchronous, sequential transfers are generally ideal with respect to the desired values of these factors and therefore to effective bandwidth utilization. The only difficulty with this type of processor is that a rela-

*During the past 18 months of operation, Project Genie's modified SDS 930 has experienced no memory failures.

tively high cost recovery procedure must be invoked whenever a transfer is deferred so long that the synchronism of the transfer is destroyed. On the other hand, processors making very high rate synchronous transfers, with their short transfer intervals, and processors making high frequency references to quasi-random addresses, such as CPU's, are far from ideal with respect to these factors.

From the above discussion, it should be evident that processors which distribute their references evenly over the memory modules and which have transfer intervals of several memory cycles provide a potential for high utilization of the available memory and memory bus bandwidths. However, this potential can be realized only when the references are made at optimum times. To insure this timing of references would require the use of some extremely clever logic which would consider not only the requests presented at a given moment, but also requests to be presented within the next several memory cycles. A fairly simple alternative to this anticipatory logic which produces near-optimum results for systems of reasonable complexity is described in the following section.

Description of request priority assignment

Given a computer system such as the one shown in Figure 1, it is common practice to use priorities assigned to the various processors and busses to resolve conflicting requests for a memory module or a memory bus.[5] These priorities are fixed and generally assigned primarily on the basis of two considerations:

1. The lower the priority assigned to a bus or processor, the lower the amount of interference to other processor-memory intercommunications caused by requests associated with this bus or processor.
2. The cost associated with the disruption of a synchronous data transfer, such as from a drum processor to memory, is substantial.

Consequently, the I/O bus (or busses) are generally assigned a priority higher than that of the CPU bus; and the processors handling high transfer rate, synchronous devices are given priorities higher than those handling lower transfer rate or asynchronous transfers. More precisely, the priority assignment is generally made by assigning to each processor the lowest priority which will insure that, under worst conditions, no high cost disruption of data flow or loss of data will occur.

The following examples will illustrate the inadequacy of this type of priority assignment. First, consider a case in which an I/O processor handling a medium-rate-transfer attempts to reference a particular memory module simultaneously with the CPU. Usually, the I/O processor can step aside and allow the CPU to make its reference without any difficulty since it can simply repeat the memory request at the next memory cycle. However, if the fixed priority assignment described above were used, the I/O processor would usurp the memory module and thereby degrade the performance of the CPU, since the I/O processor would have been assigned a priority higher than that of the CPU to guard against the infrequent instances in which it cannot further defer a memory reference. Second, consider the case of the CRT display processor which refreshes the CRT image. Generally, this processor can defer a particular memory reference for an appreciable time without unduly degrading the performance of the display. However, this processor is commonly connected to the same bus as is the drum processor; and this bus is required by the presence of the drum processor to have a high priority. Therefore, the requests of the display processor outrank those of the CPU even though the display processor can conveniently defer its memory reference. The consequence of these inequities is either that the utilization of the memory and memory bus bandwidths is limited, or the performance of one or more processors is severely degraded.

These examples and the discussion above suggest that priorities should be assigned to individual requests rather than to processors and busses, and that each processor should be given a selection of priorities for assignment to a given request. With a priority mechanism having this dynamic priority assignment capability, a processor can choose for each request a priority which is commensurate with 1) the cost to be incurred by the rejection of the request at that particular moment, 2) the probability that a request with a given priority will be rejected at that particular moment, and 3) the general desirability of making low priority, minimal interference requests. The exact costs and probabilities are, of course, difficult to ascertain; however, in most instances, they can be adequately approximated. Moreover, in every instance they can be assumed such that the resulting priority assignments are at least as good as the fixed assignments to processors currently used.

Numerous examples of improved system performance realized by the use of dynamic priority assignment are given in the next section. At this point, the following two examples should illustrate the advantages and some of the ramifications of this type of priority assignment. First, consider the priorities assigned to requests by an I/O processor handling a synchronous transfer. If it is assumed that the highest priority available to this processor is higher than that available to any other processor, then the

probability of having a request of this highest priority accepted is one. Therefore, this processor will commence a sequence of requests of relatively low priority at the beginning of each of its transfer intervals and continue this sequence until a request is accepted or until only *one* memory cycle remains in its transfer interval. Only when the entire sequence of low priority requests is rejected must the processor resort to making a high priority request. Second, consider the display processor mentioned above. The cost of deferring one of its references for a few memory cycles past the end of its transfer interval is usually not very large. Therefore, this processor might commence a transfer interval with a sequence of requests of very low priority, change to a medium priority near the end of its transfer interval, and change to a relatively high priority sometime following its transfer interval. Thus, if the frequency of requests made by this processor is not too high, the probability of its interfering with requests of other processors is extremely low.

Processor-memory configurations

Results of simulations of several processor-memory configurations and I/O processor buffering mechanisms are presented in this section. The characteristics of the memory modules and the basic memory bussing mechanism are the same for all situations and are similar to those of the system illustrated in Figure 1. More precisely, the memory modules are all identical, having an access time (the time required to read a word from memory and make it available to a processor) of .8 usec and a cycle time of 1.75 usec. As before, the several memory modules are synchronized, and a processor making a memory reference is logically connected to the memory module via its memory bus during the entire 1.75 usec memory cycle. The memory addresses are interleaved over all of the memory modules in each of the configurations considered.

The CPU simulated is the SDS 930 as modified by Project Genie.[7] This is a word-oriented processor having a single instruction per word and a single operand address per instruction. It has one index register and provides for indefinite levels of indirect addressing with pre-indexing at each level. The instruction repertoire includes one conditional jump instruction, several conditional skip instructions, numerous central register manipulation instructions (not requiring a memory reference for an operand), some memory-to-memory instructions (e.g., increment contents of the specified memory location) which require two consecutive references to the same memory location and a full complement of arithmetic and logic instructions.

Most instructions require two memory cycles for execution: ½ cycle for indexing, ½ cycle for fetching the operand, and one cycle for processing. The operand is regenerated in memory during the first ½ cycle of processing and the next instruction is fetched during the last ½ cycle. The instruction is regenerated during the indexing ½ cycle. A pertinent peculiarity of this processor is that all store instructions require three cycles; no memory reference is made during the middle cycle.

Number of Memory Modules	Probability of no memory reference at next cycle	Probability of reference to same module at next cycle	Probability of reference to same module at next two cycles	Probability of reference to same module at next three cycles	Probability of reference to same module at next four cycles	Probability of reference to same module at next five cycles
2	.091	.454	.166	.054	.012	.004
4	.091	.230	.040	.010	.002	.001
8	.091	.117	.008	.001	.000	.000
16	.091	.070	.002	.000	.000	.000

Table I—Probabilities of the CPU referencing the same memory module at consecutive memory cycles (following a reference to a particular module)

As might be expected, simulations of this CPU show that the distribution of memory references over the memory modules is uniform and quasi-random. Some probabilities of multiple consecutive references to a module are given in Table I. Two observations of these figures, which are averages over all of the modules in each configuration simulated, warrant comment: first, the approximately 9% of the cycles during which the CPU makes no memory reference result primarily from the execution of store instructions; second, the probability of two consecutive references to a module is slightly higher than might be expected. This higher value, particularly noticeable in the case of 16 modules, results primarily from the execution of those instructions which made consecutive operand references to the same memory location. The program used in the simulations was the Berkeley Time-Sharing System's macro assembler. Although several other programs including a text editor and a list processing interpreter were also simulated, the results obtained from these simulations were almost identical to those obtained using the assembler and therefore are not presented.

With this background, consider the configuration illustrated in Figure 2, which is similar to that in Figure 1. As mentioned previously, the two memory busses B1 and B2 provide for simultaneous transfers, one to the CPU and one to an I/O processor. There are two types of I/O processors shown: a drum pro-

cessor which transfers data at an average rate of 1 word every 8 usec, and four independent CRT display processors, each of which fetches data at an approximate rate of 1 word every 20 used. More precisely, display processors 1 through 4 require for 100% performance a word every 20.1, 19.9, 20.2, and 19.8 usec respectively. Furthermore, the storage capability of these processors is limited in that they cannot accept a word during the first 10 usec portion of these periods.

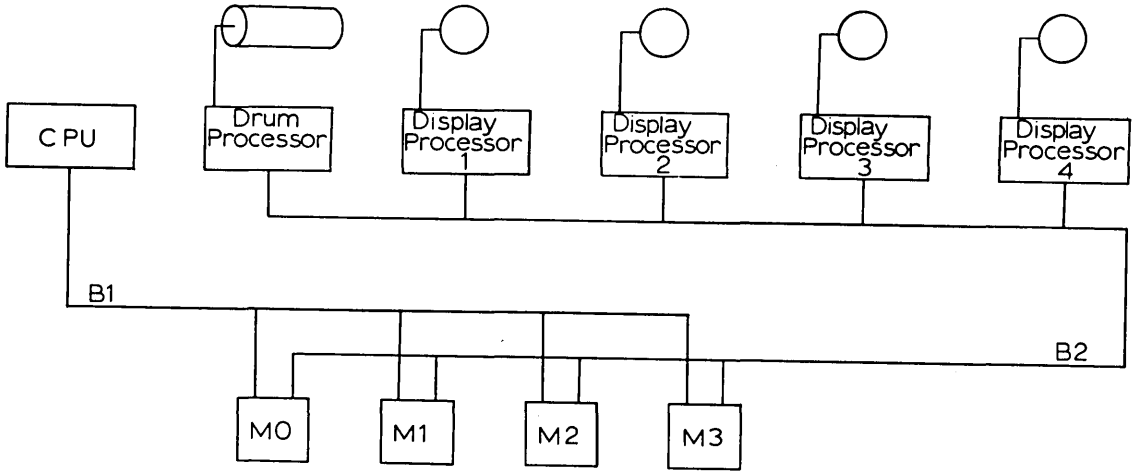


Figure 2 – Four display-processor configuration

Each of the I/O processors has two priorities (high and low) available for assignment to its requests. These priorities, which are used to determine access to both the I/O bus and the memory module, are assigned by each I/O processor in a fixed sequence of N low priority requests followed by as many high priority requests as necessary. The value of N chosen for each processor is determined by the length of the processors transfer interval, the cost of failing to make a memory reference within this interval, and the probability of having one of its high priority requests accepted. This probability is dependent, in turn, on the relative priorities which are shown in Figure 3. Because the cost of a failure by the drum processor to make a memory reference within its transfer interval is substantial, the value of N for this processor is conservatively chosen so that its performance is assured to be 1.00.

Some results of simulations of this configuration are given in Table II, where the tabulated parameters have the following definitions:

CPU performance – the ratio of instruction execution rate in the environment simulated to the rate in an environment free of memory interference.

Display processor and drum processor performance – the ratio of the number of words transferred in the environment simulated to the number transferred in an environment free of memory interference.

Memory bandwidth utilization – the ratio of the number of memory references made in a given period to the number of references possible in that period (i.e., the product of the number of memory modules and the number of memory cycles in the period).

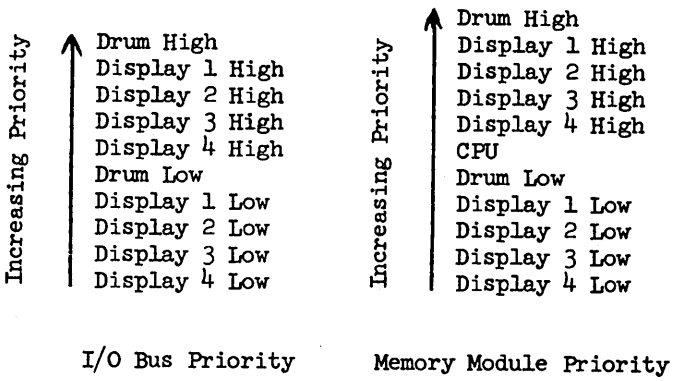


Figure 3 – Relative priority of requests

These results show the improved CPU performance and memory bandwidth utilization obtainable using dynamic priority assignment with a very simple priority assignment algorithm. More specifically, they show that in the configuration including only two processors, a CPU and a drum processor, the use of dynamic priority assignment and the memory

Number of Memory Modules	Number of I/O Busses	Number of Drum Processors	Number of Display Processors	Number of Low Priority Requests by Drum Processor	Number of Low Priority Requests by Display Processor	CPU Performance	Display Processor Performances Proc1	Proc2	Proc3	Proc4	Memory Bandwidth Utilization
2	1	1	0	0	-	.[illegible]	-	-	-	-	.[illegible]
2	1	1	0	[illegible]	-	.[illegible]	-	-	-	-	.[illegible]
4	1	1	0	0	-	.964	-	-	-	-	.[illegible]
4	1	1	0	[illegible]	-	.[illegible]	-	-	-	-	.[illegible]
2	1	0	[illegible]	-	0	.[illegible]	1.000	1.000	1.000	1.000	.[illegible]
2	1	0	[illegible]	-	[illegible]	.[illegible]	.999	.999	.999	.999	.[illegible]
2	1	0	[illegible]	-	[illegible]	1.000	.999	.997	.997	.997	.630
4	1	0	[illegible]	-	0	.919	1.000	1.000	1.000	1.000	.[illegible]
4	1	0	[illegible]	-	[illegible]	.[illegible]	1.000	1.000	.999	.999	.[illegible]
2	1	1	[illegible]	0	0	.[illegible]	1.000	.999	1.000	.999	.616
2	1	1	[illegible]	[illegible]	0	.[illegible]	1.000	1.000	1.000	1.000	.666
2	1	1	[illegible]	0	[illegible]	.[illegible]	.998	.993	.996	.986	.684
2	1	1	[illegible]	[illegible]	[illegible]	.9[illegible]0	.998	.992	.997	.989	.[illegible]
2	1	1	[illegible]	[illegible]	[illegible]	.[illegible]	.994	.984	.986	.963	.[illegible]
4	1	1	[illegible]	0	0	.[illegible]	1.000	1.000	.999	.999	.319
4	1	1	[illegible]	[illegible]	[illegible]	.[illegible]	.999	.999	.999	.998	.349

Table II—Processor performances and memory bandwidth utilization for the configuration in Figure 2

bandwidth of two memory modules provides better CPU performance and a much greater memory bandwidth utilization than that provided by twice this memory bandwidth (four memory modules) and a fixed priority assignment. Also, the addition of the four display processors causes little degradation to CPU performance. In fact, if .96 display processor performance is acceptable, this addition of display processors can be made at no loss in CPU performance by simply making all display processor requests at low priority. This result substantiates a previous statement regarding the ideality of processors which 1) distribute their memory references over the modules, 2) can anticipate the need of a reference, 3) can tolerate an occasional small delay in making a reference, and 4) require references at a moderate rate.

The results presented in Table II verify that the efficiency of a computer system can be significantly increased by assigning priorities to requests rather than to processors and by having the processors make a sequence of requests with appropriate priorities for each word to be transferred. It should be observed, however, that in each case considered the aggregate transfer rate of the I/O processors sharing the I/O bus is well below the bandwidth of the bus. In fact, the case in which 73.5% of the memory bandwidth was used, only approximately 56% of the I/O bus bandwidth was used for data transfers. Most of the remaining 44% was used for low priority requests which tied up the bus even though they were rejected at the memory module. Since an excess of bus bandwidth over transfer rate is obviously necessary for the effective operation of the dynamic priority scheme, an increase in processor-memory intercommunication capability beyond that of the configuration in Figure 2 requires either an increase in the bus bandwidth available to processors or a means by which more effective utilization can be made of a given bus bandwidth. Let us first consider methods of increasing the bus bandwidth available to the I/O processors.

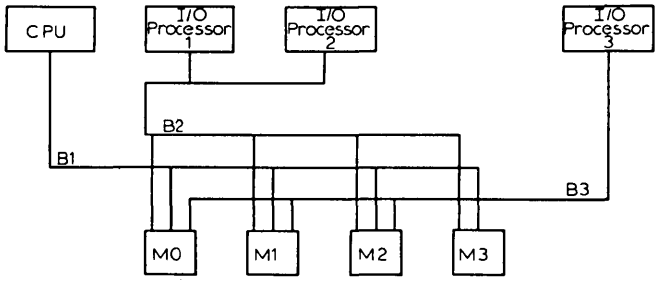


Figure 4—Increased memory buss bandwidth provided by two I/O busses

It is apparent that the maximum bandwidth available to any I/O processor in Figure 2 is limited to 1/T by the bandwidth of the I/O bus. Moreover, since the bandwidth must be shared by the several I/O processors, the actual bandwidth available to a given processor is considerably less than this limit. As illustrated in Figure 4, providing two I/O busses doubles the bandwidth available to the I/O processors while retaining the absolute limit of 1/T for any one processor. To increase this latter limit, two (or more) busses can be provided a given processor. This can be done either by providing the processor with two busses, each of which connects to all of the memory modules, or by providing one bus which connects to the even-numbered modules and one which connects to the odd-numbered modules as shown for I/O processor 3 in Figure 5. The second alternative requires less logic to implement and, since alternate busses are used in referencing sequential addresses, it is as effective for processors which reference sequential addresses as the first alternative. Therefore this second alternative, which can be extended in an obvious manner to four busses (provided the number of modules is an integer multiple of four), will be discussed exclusively in the remainder of this paper.

To illustrate the value of increased bus bandwidth to the processors, simulation results for the configuration shown in Figure 6 are given in Figure 7. The drum processor, which transfers data at the rates indicated, has a single buffer register and separate,

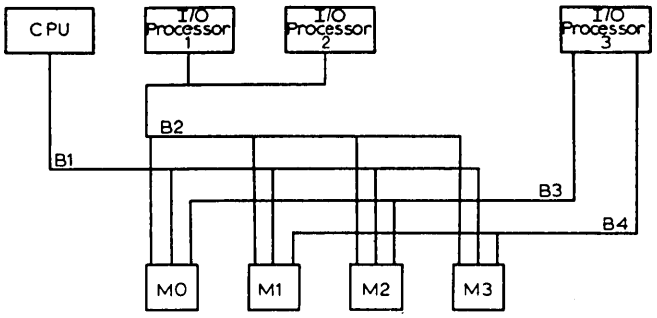


Figure 5 – Increased memory buss bandwidth provided by two busses to a single I/O processor

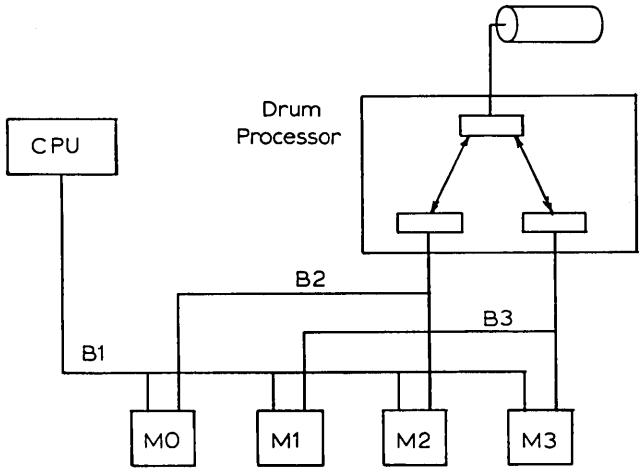


Figure 6 – Drum processor with single buffer registers

identical, request logic for each bus to which it is attached (in addition to a single buffer associated with the drum write amplifiers and read amplifiers.) The request logic begins making low priority requests when its associated buffer register becomes ready (full on drum-to-memory transfers and empty on memory-to-drum transfers). It continues to make low priority requests until either a request is granted or only one memory cycle remains in the transfer interval. When the latter situation occurs, a request having a priority higher than that of CPU requests is made, thus insuring that the transfer will be made within the transfer interval. For comparison, several results are given for request logic which always makes high priority requests. These results are presented in tabular form together with memory bandwidth utilizations in Table III.

As was anticipated, dynamic priority assignment is effective only when the data transfer rate of the drum processor is considerably less than the bus bandwidth which is available to it. However, the results for the two memory module configuration with two memory busses to the drum processor suggest that dynamic priority assignment is only slightly effective when there is a two to one ratio of bus bandwidth to transfer rate. This is not generally true; this anomaly is caused by a tendency for the CPU in this particular case to get into phase with the drum processor, alternating memory references to each of the two memory modules. This phenomenon is illustrated in Figure 8 for a sequence of typical two cycle instructions. As shown in this figure, when the drum processor makes only high priority requests (one request every other cycle on each of its two busses), the execution of each of the two cycle instructions requires three cycles, regardless of which module contains the operand. Furthermore, when each high priority request made by the drum processor is replaced by a low priority request followed, if necessary, by a high priority request, then some of these low priority requests will be accepted—thereby causing the associated references to occur one memory cycle earlier, but not changing the fact that each of the two cycle instructions requires three cycles for execution. It should be observed, however, that even in this unfavorable situation, the two-module, two I/O bus configuration with dynamic priority yields a higher CPU performance and memory bandwidth utilization at the lower drum processor transfer rates than a four-module, two I/O bus configuration with fixed priority.

To improve the processor performance in the situation just described, and to generally increase the effectiveness of a given bus bandwidth, the transfer intervals of I/O processors must be increased. The processor shown in Figure 9, which is similar to the one described above except that it has one additional buffer register inserted into the path of the data flow to each bus, demonstrates one method of achieving this goal. To illustrate the function of this buffer, consider a drum with a data transfer rate of 1 word every 2 usec. Assume that a memory-to-drum transfer is in progress and, at a time designated t_0, BR1 is full (with contents β) and BR2 (with contents α) is in the process of being transferred to D, a process which consumes 100 nsec. Then at $t_0 + 100$ nsec BR2 is empty, and a BR1 to BR2 transfer is initiated; at $t_0 + 200$ nsec the transfer is completed and BR1 is empty. Since the drum receives only every other word from BR2, β is not required by the drum until $t_0 + 4000$ nsec and ∂, the next word to be fetched into BR1, is not required until $t_0 + 8000$ nsec. Therefore,

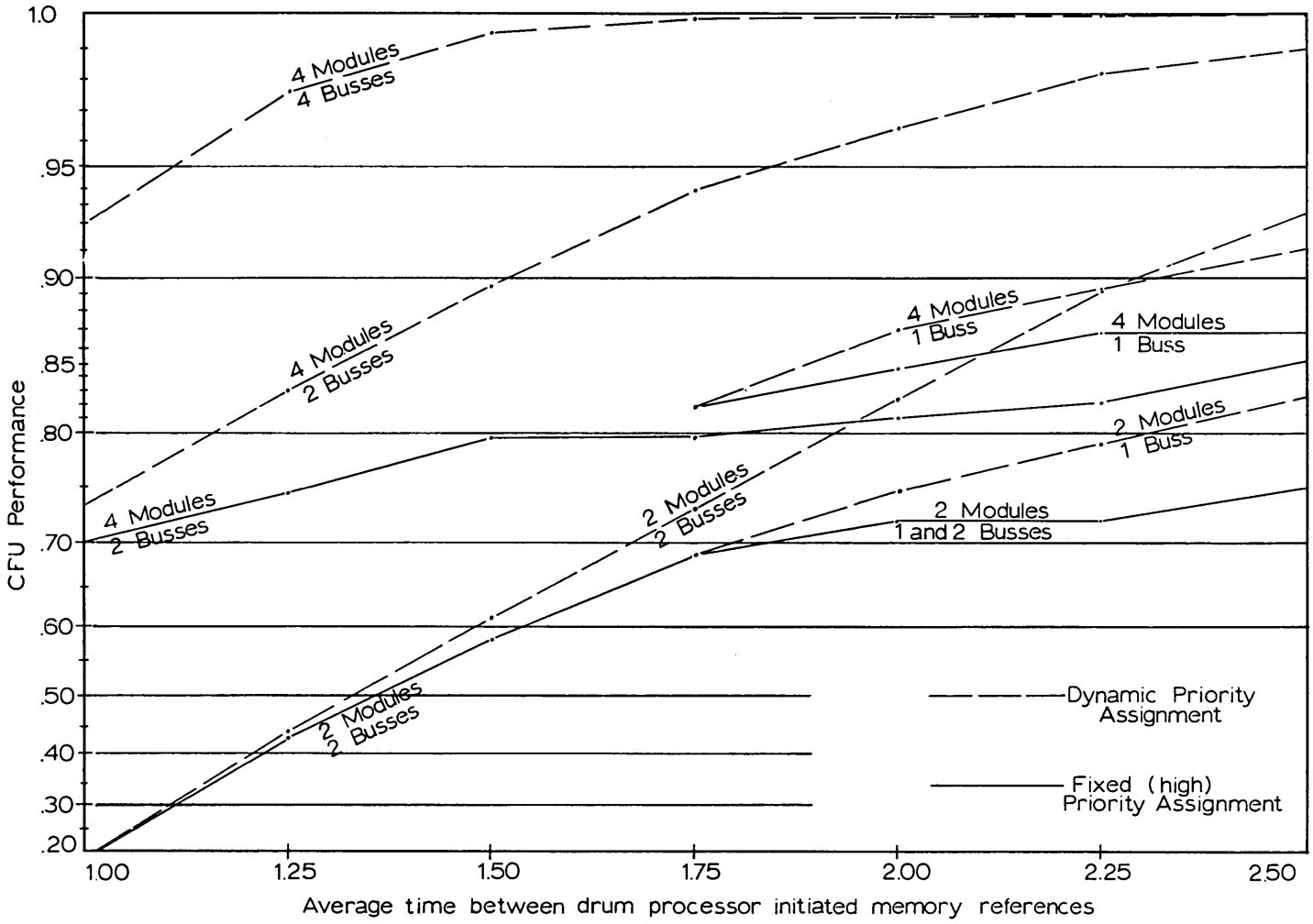


Figure 7—CPU performance for single buffered drum processor (for 2 and 4 memory modules and 1,2, and 4 memory busses to the drum processor)

Number of Memory Modules	Number of Memory Busses to Drum Processor	Average time between drum processor initiated memory references: 1.00 usec. CPU*	1.00 usec. Mem.**	1.25 usec. CPU	1.25 usec. Mem.	1.50 usec. CPU	1.50 usec. Mem.	1.75 usec. CPU	1.75 usec. Mem.	2.00 usec. CPU	2.00 usec. Mem.	2.25 usec. CPU	2.25 usec. Mem.	2.50 usec. CPU	2.50 usec. Mem.
2	1							.690	.815	.774	.790	.836	.770	.881	.752
2	2	.230	.980	.553	.952	.759	.930	.893	.907	.953	.872	.980	.836	.993	.802
4	1							.819	.437	.900	.424	.946	.410	.966	.395
4	2	.846	.630	.956	.568	.990	.517	.997	.477	.998	.446	.999	.422	.999	.403
4	4	.995	.666	.999	.578	.999	.520	1.000	.478	1.000	.447	1.000	.422	1.000	.403

*CPU performance.
**Memory bandwidth utilization.

Table III—CPU performance and memory bandwidth utilization for single buffered drum processor (Figure 6)

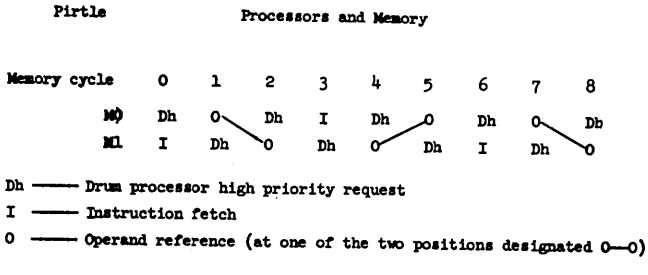


Figure 8a—Memory reference sequence when drum processor makes high priority requests only

there is a 7.7 usec period in which to fetch ∂ into BR1. This period is greater than the time required for four memory cycles; therefore, the processor can make two or possibly three low priority requests

Memory cycle	0	1	2	3	4	5	6	7	8
M0	Dh	O/Dl	Dh/N	I	Dh	O/Dl	Dh/N	O/Dl	Dh/N
M1	I	Dh	O/Dl	Dh/N	O/Dl	Dh/N	I	Dh	O/Dl

Dl —— Drum processor low priority request
N —— No request
O/Dl —— Dl if no operand reference
Dh/N —— No request if no drum processor request

Figure 8b—Memory reference sequence when drum processor makes a low-high priority sequence of requests

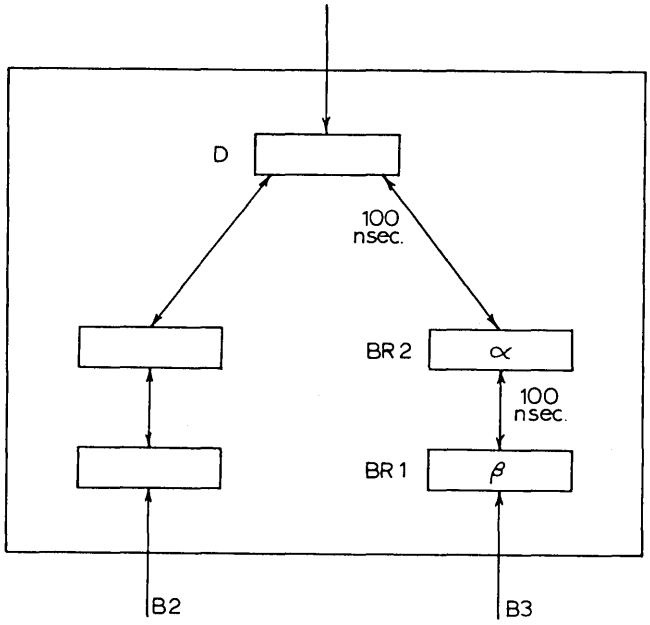


Figure 9—Drum processor with double buffer registers

before resorting to a high priority request, instead of the one low priority request possible with the processor having a single buffer register. It should be apparent that the second buffer register allows the processor to take advantage of the acceptance of a low priority request to lengthen its subsequent transfer interval.

CPU performances obtained with a drum processor having two buffer registers are given in Figure 10 and tabulated along with memory bandwidth utilizations in Table IV. The CPU performances presented in this figure are uniformly higher than those obtained for I/O processors having single buffer registers. In fact, two combinations of memory and bus bandwidths yield very good results: the two-module, two I/O bus configuration yields a CPU performance in excess of .95 and a memory bandwidth utilization in excess of .80 for drum processor transfer rates between 1 word per 2.00 usec and 1 word per 2.50 usec; the four-module, two I/O bus configuration yields a CPU performance in excess of .95 and a memory bandwidth utilization in excess of .50 for drum processor transfer rates greater than the bandwidth of a single module. Also, the four-module, four I/O bus configuration yields an impressive .99 CPU performance for a drum processor transfer rate approaching twice the bandwidth of a single module.

The effect on CPU performance and memory bandwidth utilization caused by the addition of a disc processor to a configuration similar to that in Figure 6 is given in Table V. These results are for drum and disc processors which make memory references (on the average) once every 2 usec and 18 usec respectively. Both processors have double buffer registers and, as shown in Figure 11, they have separate memory busses. The disc processor has request logic which makes low priority requests until a request is accepted or until only three memory cycles remain in the transfer interval; in the latter event, high priority requests are made until one is accepted. Since the drum processor, which has the highest request priority at its disposal (Figure 12), will not make more than two consecutive high priority requests to any particular memory module (at the transfer rate indicated), the disc processor is assured of a memory reference in each transfer interval. A comparison of these results with those given previously shows that in most cases the addition of the disc processor causes only a very small degradation of CPU performance. Although this disc processor must make a memory reference each transfer interval, its relatively low transfer rate and its buffer-register-extended transfer interval make it a near ideal processor. This is evinced by the near optimum result of a CPU performance of .94 and a memory utilization of almost .91 for a two-module configuration.

CONCLUSION

It has been shown that the assignment of priorities to requests, rather than to processors and busses, can be very effective when used in conjunction with memory systems which provide ample memory bus bandwidth to the processors. Although an excess of bus bandwidth over processor transfer rate must be provided, processors with appropriate data buffering mechanisms can operate effectively with transfer rate to bus bandwidth ratios exceeding three-fourths. More importantly, in optimum configurations the several processors, can operate at very high per-

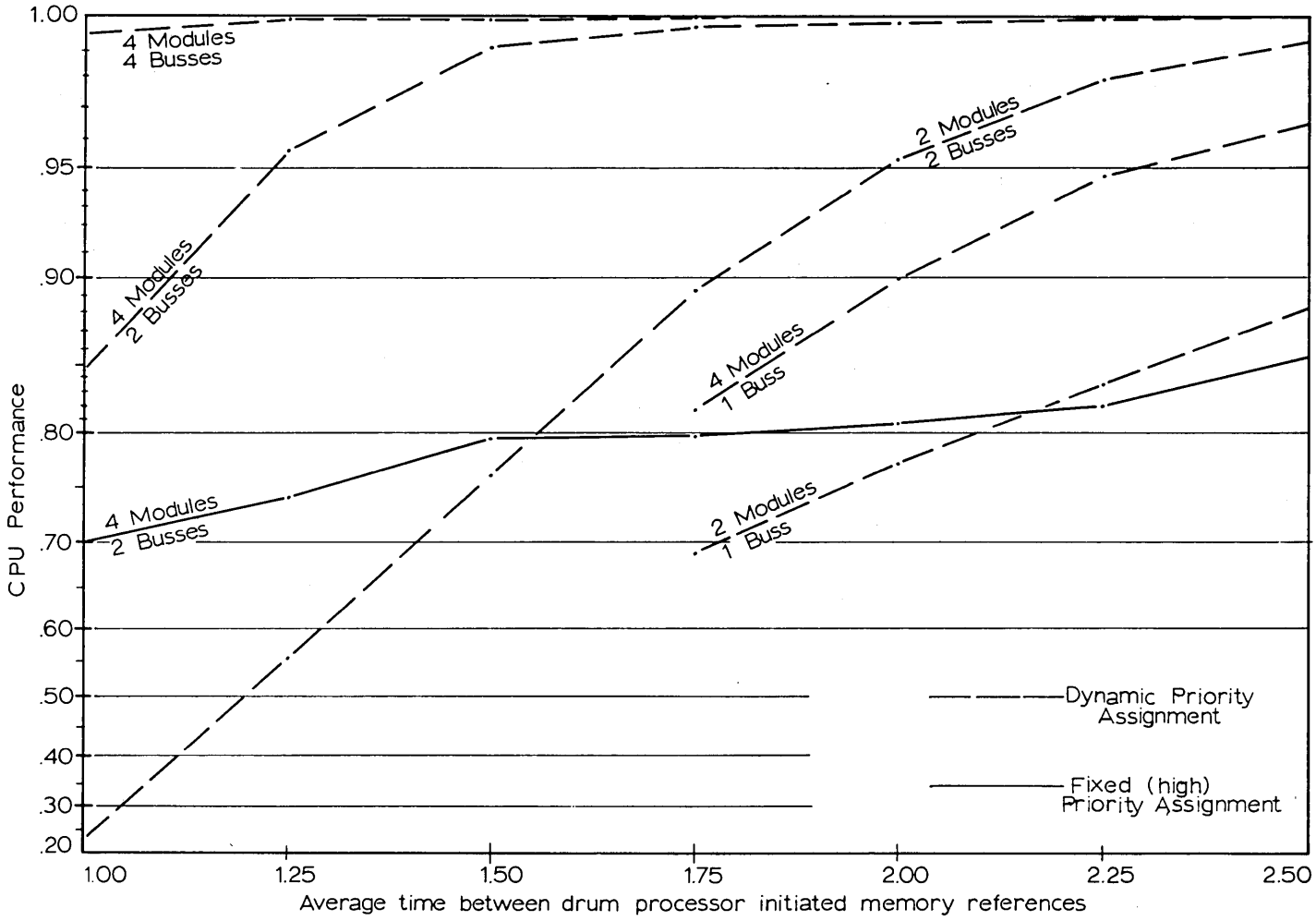


Figure 10—CPU performance for double buffered drum processor (for 2 and 4 memory modules and 1,2, and 4 memory busses to the drum processor).

Number of Memory Modules	Number of Memory Busses to Drum Processor	Dynamic Priority Assignment Employed?	Average time between drum processor initiated memory references: 1.00 usec. CPU*	1.00 usec. Mem.**	1.25 usec. CPU	1.25 usec. Mem.	1.50 usec. CPU	1.50 usec. Mem.	1.75 usec. CPU	1.75 usec. Mem.	2.00 usec. CPU	2.00 usec. Mem.	2.25 usec. CPU	2.25 usec. Mem.	2.50 usec. CPU	2.50 usec. Mem.
2	1	yes							.690	.815	.744	.777	.786	.747	.806	.777
2	1	no							.690	.815	.717	.764	.716	.715	.753	.694
2	2	yes	.184	.959	.441	.901	.610	.861	.731	.833	.823	.813	.892	.796	.930	.774
2	2	no	.184	.959	.430	.896	.581	.848	.690	.815	.717	.764	.716	.715	.753	.694
4	1	yes							.818	.437	.870	.417	.897	.399	.912	.383
4	1	no							.818	.437	.847	.413	.867	.392	.867	.373
4	2	yes	.732	.604	.830	.539	.895	.496	.939	.464	.964	.439	.981	.418	.990	.401
4	2	no	.699	.597	.743	.519	.796	.473	.798	.432	.810	.404	.821	.381	.865	.370
4	4	yes	.924	.648	.976	.573	.994	.518	.998	.478	.999	.447	.999	.422	1.000	.403

*CPU performance.
**Memory bandwidth utilization.

Table IV—CPU performance and memory bandwidth utilization for double buffered drum processor (Figure 9)

Number of Memory Modules	Number of Busses to Drum Processor	Number of Busses to Disc Processor	CPU Performance	Memory Bandwidth Utilization
2	1	1	.772	.837
2	2	1	.931	.909
4	1	1	.898	.447
4	2	1	.998	.470
4	4	1	1.000	.470

Table V—CPU performance and memory bandwidth utilization for double buffered drum and disc processor averaging one memory reference every 2 usec and 18 usec respectively

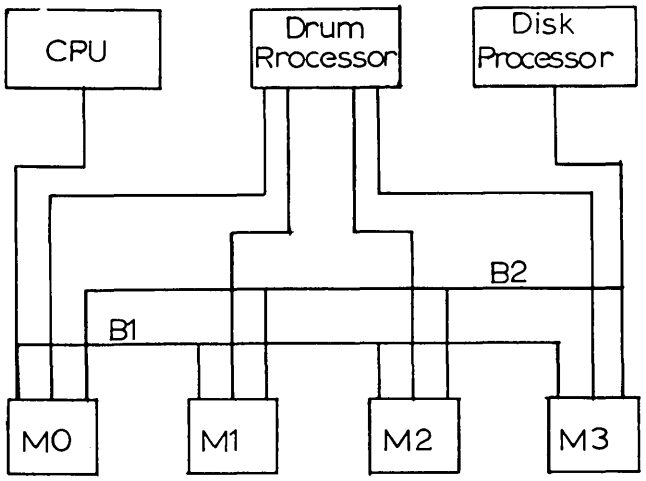


Figure 11 – Drum processor and disc processor (with double buffer registers)

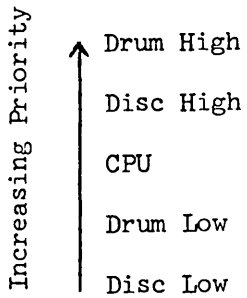


Figure 12 – Memory request priorities for CPU, drum and disc processor configuration

formance figures while using more than three-fourths of the memory bandwidth.

It should be noted, however, that the results presented in this paper were obtained from simulations of computer systems having a relatively small number of high-transfer-rate processors and having synchronous memory systems. As a result it was possible to make the logic used in these systems to determine the priority of a given request relatively simple. Systems having several high-transfer-rate processors, however, must provide mechanisms by which priorities can be determined on the basis of other requests made at the same time by other processors. For example, in the systems described, the single high-transfer-rate processor (drum processor) makes low priority requests until there is only one memory cycle remaining in which to make the reference. It can wait this long because there is no processor which can make a request of higher priority. If no additional means of assigning and interpreting priorities is provided, then one of the drum processors in a four drum system would have to start making high priority requests when there were four memory cycles remaining in its transfer interval. This is obviously not an optimum procedure. A more reasonable method is to assign a priority which reflects precisely the number of cyles remaining before the reference must be made. Logic at each memory module can then interpret the priority of requests from a more englightened viewpoint, considering the likelihood of congestion at the module within a few cycles. For example, it a request from a CPU is received by a module simultaneously with two drum processor requests, the CPU request should be accepted unless the drum requests indicate that 1) both drums require a reference within two memory cycles or 2) one drum requires a reference immediately. A simple scheme which is effective in systems with only two or three high-transfer-rate I/O processors provides a "warning" priority which is employed on the one or two cycles preceding high priority requests. At each module, a single warning priority request is interpreted as a low priority request whereas multiple (simultaneous) warning-priority requests are interpreted as high priority requests. This interpretation assists in allowing a drum processor to make several low priority requests while precluding the occurrence of multiple high priority requests at a given module which cannot be accommodated.

An additional complication is introduced if the memory system is asynchronous. Such a system allows a processor to make a memory request at an arbitrary time, without regard for memory cycle times. This freedom provides several advantages, such as allowing the CPU to make an operand reference immediately upon determining its effective address without waiting for the next memory cycle. However, this type of memory is not amenable to dynamic priority assignment as described, because most requests in this type of system are granted on a first-come-first-served basis. Only when multiple requests are presented to a module (or bus) simultaneously is a decision based on priority. Work on a memory implementation which will provide the advantages of asynchronous systems and dynamic priority assignment, plus other advantages, is currently being pursued by the author; preliminary results from this work are very encouraging.

ACKNOWLEDGMENTS

For the simulation results presented, I am indebted to Mr. Eric Ha who wrote the simulation programs

and to the programming staff of Project Genie for the Berkeley Time-Sharing System on which these programs were run.

REFERENCES

1 C T CASALE
Planning the 3600
Proceedings of the Fall Joint Computer Conference 1962 pp 73-85

2 G M AMDAHL
Engineering aspects of large, high-speed computer design; Part II – logical organization
IBM Technical Report TROO.1227 December 18 1964

3 M KENRO K NAKAZAWA
Very high speed and serial-parallel computers HITAC 5020 and 5020E
Proceedings of the Fall Joint computer conference 1964 pp 187-203

4 J P ANDERSON et al
D825 – A multiple-computer system for command and control
Proceedings of the Fall Joint Computer Conference 1962 00 86-96

5 *System description Univac 1108 multi-processing system*
Univac Reference manual, UP-4046 Rev 1

6 M J FLYNN
Very high-speed computing system
Proceedings of the IEEE Vol 54 No 12 December 1966 pp 1901-1909

7 W W LICHTENBERGER M W PIRTLE
A facility for experimentation in man-machine interaction
Proceedings of the fall joint computer conference 1965 pp 589-598

Stochastic computing elements and systems

by W. J. POPPELBAUM, C. AFUSO and J. W. ESCH
University of Illinois
Urbana, Illinois

INTRODUCTION

To date essentially only two fundamentally different representations of numbers have been used in electronic computers: One is the *analog representation* by a voltage or current inside a given range, the other one, the *digital representation* which maps a sequence of 0's and 1's onto a spatial or temporal sequence of voltage or current pulses. Of late, interest has arisen in the use of *random pulse sequences* as information carriers. (1), (2), (3), (4), (5), (6). It turns out that *the use of random pulse sequences leads to the use of digital ANDs or ORs for the fundamental operations of multiplication and summation* and therefore to a very considerable reduction in cost of the computational equipment. It is the purpose of this paper to present the theory of these random pulse sequences as well as their practical circuit implementations, and to give some systems design examples. In the final section some non-Von Neumann organizations will be discussed which are made possible by the use of stochastic computing elements.

In the course of developing the techniques in question our initial Random Pulse Sequence System (RPS) was replaced by the Synchronous Random Pulse Sequence System (SRPS). In the former case a sequence of standardized pulses represents the variable value by its average duty cycle, with the understanding that the standardized pulses occur at entirely random times. In the SRPS system these standardized pulses occur in fixed time slots with a probability of occurrence equal to the variable value. The main advantage of the latter system is that addition and multiplication of two SRPS's in an OR or an AND leads to another SRPS without any further normalization.

Figure 1 shows a way of generating controlled RPS's and SRPS's. The principle is simply to generate white noise in a noise diode and to amplify it in a threshold device, the threshold being proportional to the variable value. Reshaping of the output of this threshold differential amplifier leads to an appropriate RPS. For the production of an SRPS it is only necessary to set a flip-flop into the 0 state by the above RPS and to clear this flip-flop to 1 by a clock having a period considerably smaller than the average period of the RPS. The pulse transmitted through a capacitor upon resetting the flip-flop (if it had been set by the RPS during the preceding clock period) can now be reshaped in order to obtain an SRPS.

Figure 2 shows the inputs and outputs of an AND and an OR receiving as their input two SRPS's. It is easily seen that the average frequency of the outputs is respectively proportional to the sum or the product of the average input frequencies. It should be noted that certain difficulties (coincident pulses) arise in the case of the OR circuit when the incoming sequences are not mutually exclusive. This case can be easily eliminated by slightly more sophisticated design discussed in Section 4. A more detailed discussion of the mapping of numbers onto RPS's and SRPS's will be found in Section 2.

It is now clear that since adders and multipliers have the simple form of ORs and ANDs the production of very complicated linear combinations of numbers can be done with great ease and at low cost. The principal application of stochastic computing elements should therefore lie in those areas where such linear combinations are of use. Happily enough, the series expansion of the most general functions contain such linear combinations. One of the most attractive applications is that of a General Image Transformer described in Section 6.

It should be emphasized at this point that theory shows (See Section 2) that approximately 10,000 pulses are needed in order to obtain 1% precision. This means that a 1% result must wait for 1 millisecond if the average pulse repetition frequency is 10 MHz. This precision is entirely sufficient for most control applications and certainly more than adequate in the processing of light intensities, i.e., video information. Unluckily, a tenfold increase in precision, i.e., a

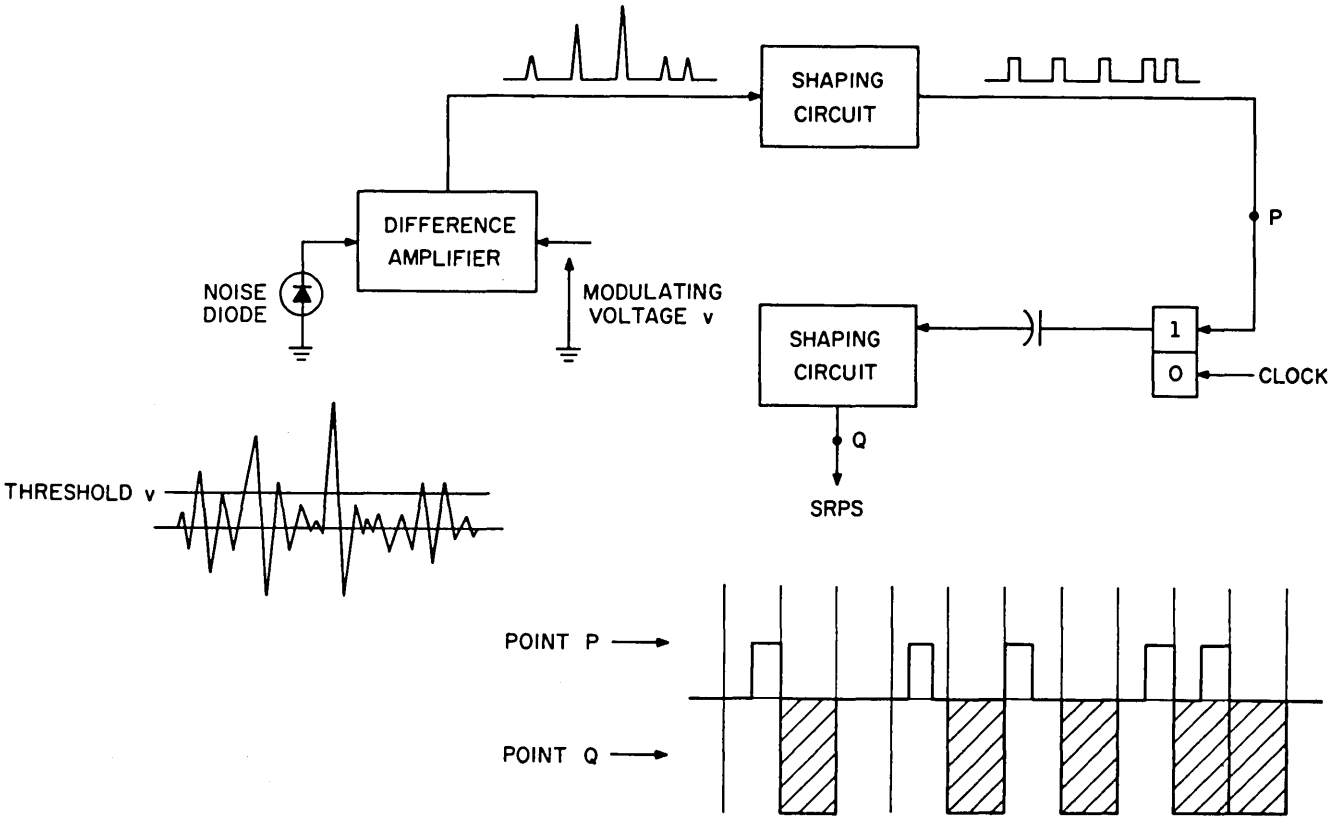


Figure 1 – Production of synchronous random pulse sequences

result known to within .1% necessitates not ten times more but 100 times more pulses. Stochastic computing elements are therefore strictly limited to calculations in which roughly .1% precision is the very upper limit. It should be emphasized that the limited precision in stochastic computing elements is inherent in the method because of the fluctuations encountered in the random process. This means that if we represent a constant value by an RPS or SRPS, the average frequency will vary slightly between adjacent time intervals. It is only when these time intervals are made long enough that the RMS value of the fluctuation becomes small enough. This still by no means excludes an entirely unlikely value for the average frequency in some very rare cases!

Mapping of numbers onto RPS's and SRPS's

In a RPS described by e(t) each pulse is of height V_0 and width τ and the frequency changes at random about an average value f: Figure 3 shows such a sequence. We can define the *average voltage V* over a time $T(T >> \tau)$ by

$$V = \frac{1}{T}\int_0^T e(t)dt \tag{2.1}$$

and the average *duty cycle x* by

$$x = \frac{V}{V_0} \tag{2.2}$$

Note that x is simply the probability at a given time of finding $e(t) = V_0$. It is easily seen that $0 \leq x \leq 1$ and that

$$f = \frac{x}{\tau} \tag{2.3}$$

In the RPS-System it is natural to represent a normalized positive number X by $x = X$. Normalization is, of course, no restriction because any computer representation demands it. The restriction $X \geq 0$ can be lifted in two ways: We can invert the pulses so as to obtain negative values of V or we can use a separate wire to carry the sign information.

As we pointed out before, the shape of pulses is destroyed by passing them through OR's and AND's. This difficulty is eliminated by timing the beginning of each pulse, should it occur, by a centralized clock. The characteristics V_0 and τ of each pulse remain unchanged. Figure 4 shows such a SRPS and it is clear that AND- or OR-operation no longer affect the

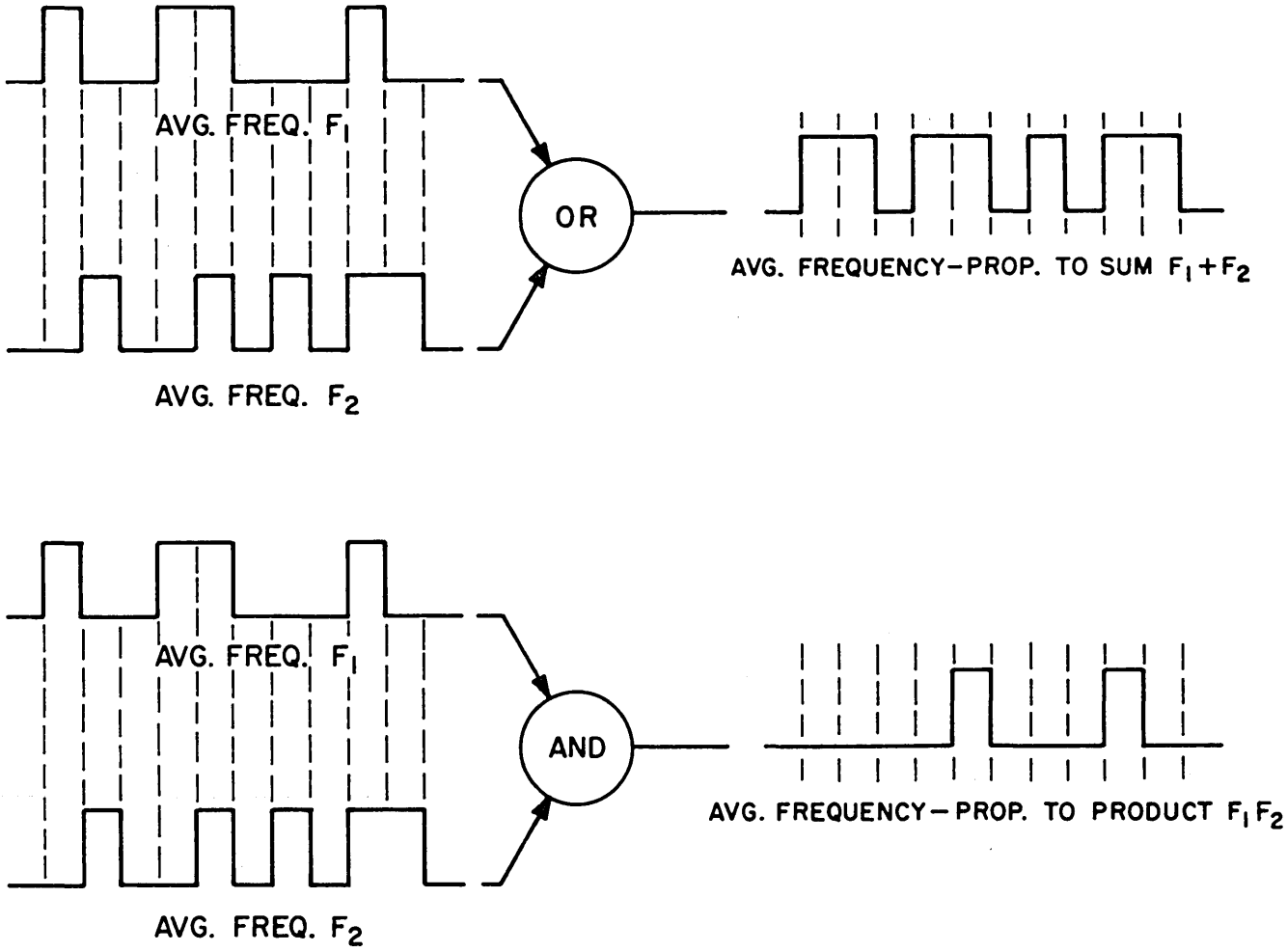


Figure 2 – Use of or's & and's for addition & multiplication in synchronous random pulse sequence system

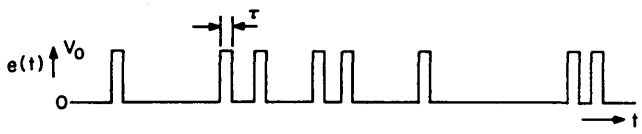


Figure 3 – A random pulse sequence

shape. It is also clear that equations (2.1)...(2.3) remain valid. Here, however, the maximum value of x is ≤ 1 because τ is usually less than the clock period T_0: It is more efficient to map X onto

$$u = \frac{f}{f_0} \text{ where } f_0 = \frac{1}{T_0} + \tag{2.4}$$

i.e., the *probability* of the occurrence of a pulse in a given time-slot. All we have done, of course, is to replace $X \rightarrow x$ by $X \rightarrow u$ with

$$u = x\left(\frac{T_0}{\tau}\right) \tag{2.5}$$

Sign questions can again be solved as above.

In practice the averaging time T is finite and all quantities in (2.1)...(2.5) show fluctuations: These, as pointed out before, are inherent in randomness and limit the dynamic accuracy of the system. For a RPS with $\tau << T$ the pulse distribution in time is described by a Poisson distribution with standard deviation given by $\sqrt{fT}$. Therefore the relative fluctuation of all variable values with respect to the average value, fT, is

$$\Delta = \frac{100}{\sqrt{fT}} \% \tag{2.6}$$

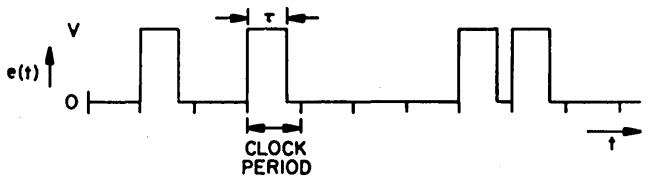


Figure 4 – A synchronous random pulse sequence

For a SRPS the probability distribution in the fixed time-slots is binomial with standard deviation given by $\sqrt{nu(1-u)}$. The relative fluctuation with respect to the average value, nu, is

$$\Delta = \frac{100}{\sqrt{n}} \cdot \sqrt{\frac{1}{u} - 1} \qquad \% \qquad (2.7)$$

where n is the number of clock pulses during T, i.e., $n = T/T_0$.

The four fundamental operations of arithmetic for RPS's

It is trivial to show that in the mapping $X \rightarrow x$ we can use OR's for addition *if we neglect the case of overlapping inputs.* The reason is, of course, that under this assumption we have simply $f = f_1 + f_2$ where f corresponds to the output and f_1 and f_2 to the inputs. This implies that

$$\frac{x}{\tau} = \frac{x_1}{\tau} + \frac{x_2}{\tau} \rightarrow x = x_1 + x_2 \qquad (3.1)$$

For multiplication we use the probabilistic interpretation of (2.2) which gives directly

$$x = x_1 \cdot x_2 \qquad (3.2)$$

for the output of an AND circuit fed by RPS's corresponding to x_1 and x_2 respectively.

Unluckily subtraction and division offer slightly more difficulty: They both necessitate the generation of an output RPS by a new RPS-generator, the latter being steered by an appropriate combination of average values. In the case of subtraction, we invert the sequence to be subtracted (i.e., form a sequence whose pulses go negative from ground) and use the difference of the average values of minuend and subtrahend to steer our generator. For division we use a layout as shown in Figure 5 in which the generator for x_3 is steered by a feedback system: When x_3 is such that x_2x_3 is on the average equal to x_1, we obviously have $x_3 = x_1/x_2$.

It is entirely possible to design a General Computing Element which performs any one of the above operations depending on a "function signal" from the outside world. Such a GCE is shown in Figures 6 and 7.

$G_1, G_2, \ldots, G_7$ are diode gates whose input impedance are practically infinite when they are OFF. For multiplication, G_1, G_3, and G_6 are ON (gate control voltage = 0) and the rest of the gates are OFF (gate control voltage = 5v if the pulse height of the R.P.S. is 5v). x_1 and x_2 are ANDed with D_1 and D_2 and the average voltage of the inversion of the product is compared with the average voltage of the inversion of the local R.P.S.G. R.P.S.G. is controlled by the comparator output such that the inputs to the comparator have the same average value.

For division, x_1/x_2, (assuming $x_1 < x_2$!) only G_2, G^3, and G_7 are ON. x_2x_3 is formed with D_2 and D_3 and the corresponding average voltage of x_2x_3 is compared with that of x_1 and the R.P.S.G. is adjusted so that $x_2x_3 = x_1$. The output, x_3, thus gives x_1/x_2.

For addition, only G_1, G_5, and G_6 are ON. Since G_3 and G_7 are OFF (5v) the AND gate formed with D_1, D_2, and D_3 gives x_1 itself. At the collector circuit of T_1 and T_7 the analog sum is formed and the R.P.S.G is adjusted so that $x_3 = x_1 + x_2$.

For subtraction, $x_1 - x_2$, assuming $x_1 \geq x_2$, only G_1, G_4, and G_6 are ON. The situation is the same as that of addition except that x_2 is fed through T_5 and T_6. Analog inversion takes place at T_6 and hence $x_3 = x_1 - x_2$.

Since R.P.S.G. employs 5 transistors and a few diodes, the whole unit employs 12 transistors and about 30 diodes. In order for the element to be able to handle negative inputs and outputs, additional sign carrying wires and sign detector circuits are needed.

The four fundamental operations of arithmetic for SRPS's

In the mapping $X \rightarrow x\left(\frac{T_0}{\tau}\right) = \frac{f}{f_0} = u$ we see without difficulty that we have again $f = f_1 + f_2$ for the output of an OR, i.e.,

$$u = u_1 + u_2 \qquad (4.1)$$

The probabilistic interpretation of u in (2.4) leads immediately to

$$u = u_1 \cdot u_2 \qquad (4.2)$$

for the output of an AND and therefore produces multiplication, this time without danger of damaging the pulse shapes.

Having in hand addition and multiplication by purely digital methods it becomes attractive to handle subtraction and division in a *purely digital fashion.* Happily enough this is possible in the case of SRPS's because of their occurrence in fixed time-slots. For subtraction we can build a simple *deletion circuit* in

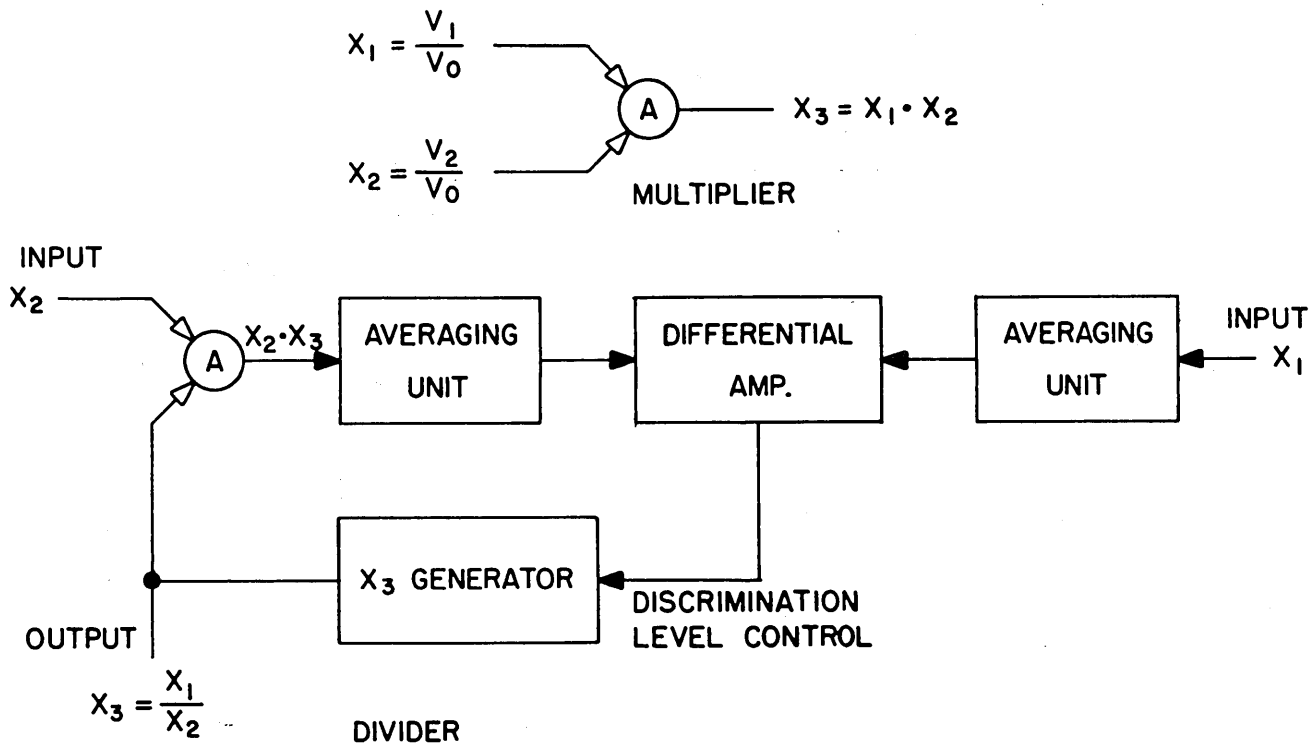


Figure 5 – Multiplier and divider in the R.P.S. system

which (for the case minuend > subtrahend) the occurrence of a subtrahend pulse leads to the deletion of the next minuend pulse. In order to obtain higher precision a small memory buffer may be incorporated so that bunching of subtrahend pulses can be taken care of. Note that the output of such a deletion circuit is again a SRPS and that obviously $f = f_1 - f_2$.

Division finally can also be handled digitally by observing that the problem X/Y can be solved for $X << 1$ and $0 \leq Y \leq 1$ by the following procedure:

1. Each time an X-pulse occurs, mark the next Y-pulse as the beginning of a "counting interval" and the Y-pulse following it as the end of a "counting interval."
2. Count all time-slots inside a "counting interval" and add 1.
3. The SRPS consisting of groups of bunched pulses in "counting intervals" represents $Z = X/Y$.

That this is effectively so, is seen by considering a very long sampling time T: in it there will be (See Figure 8)

$$n = Tf_0X \quad \text{X-pulses} \tag{4.3}$$

since $X = f/f_0$. The average period S of the Y-pulses is given by

$$S = \frac{1}{Yf_0} \tag{4.4}$$

This is by definition the length of a "counting interval" and corresponds therefore to Sf_0 pulses of the output sequence. During T we shall observe n bunches of Sf_0 pulses, i.e., the output represents a number Z such that

$$Z = \frac{\text{avg. frequency}}{f_0} = \frac{\text{\# of pulses}}{Tf_0}$$

$$= \frac{n(Sf_0)}{Tf_0} = \frac{(Tf_0X)}{(Yf_0)T} = \frac{X}{Y} \tag{4.5}$$

Again it is possible to design a GCE which, however, this time uses digital circuitry only. Figures 9-12 show a possible solution in which only the important paths are shown in a GCE consisting of

1. An UP-DOWN Counter.
2. A J-K Flipflop.

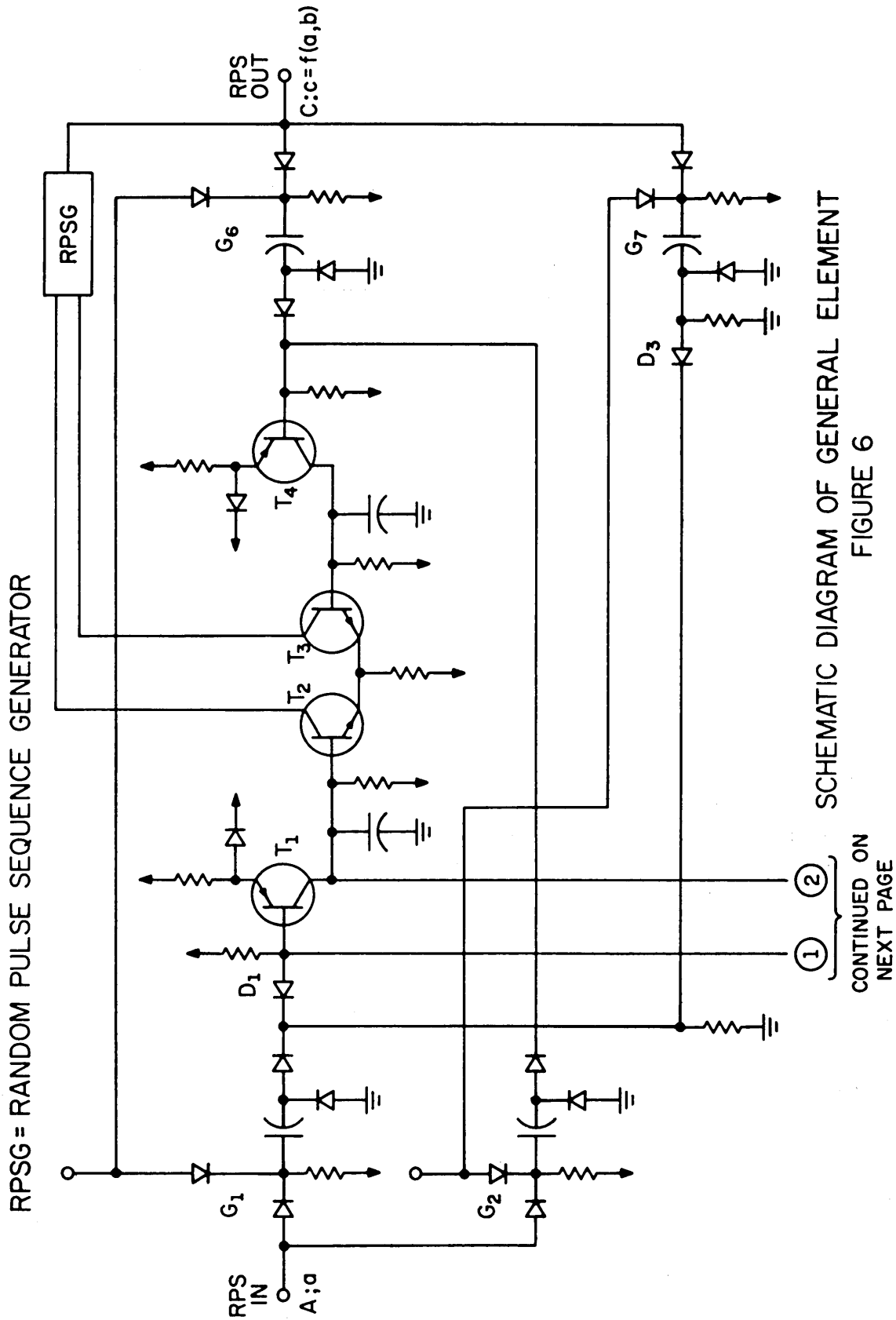


Figure 6 – Schematic diagram of general element

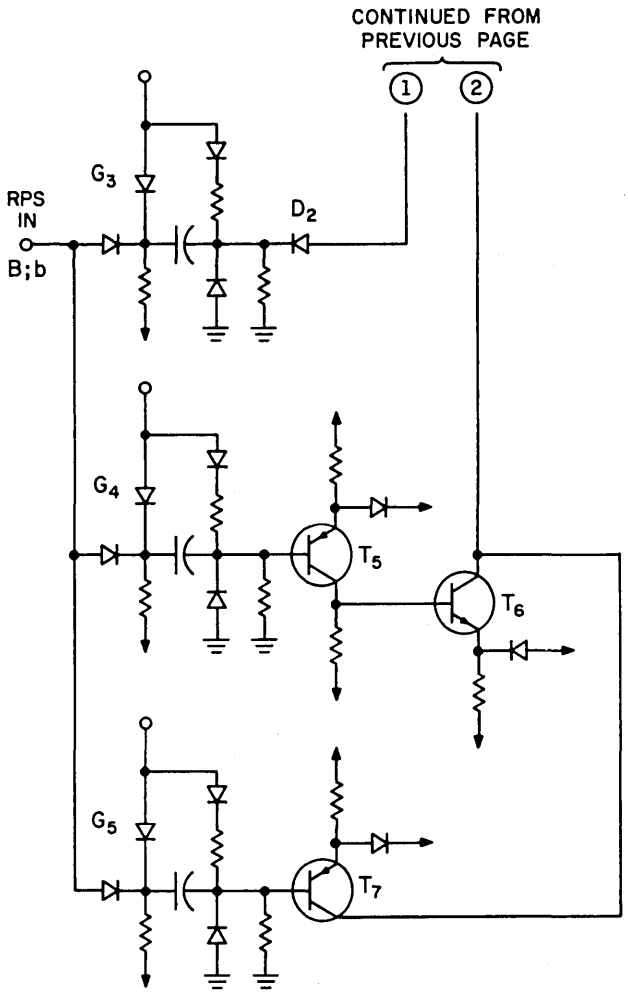


Figure 7 – Schematic diagram of general element

3. Digital Steering Circuitry.
4. Feedback Paths W and Z.

The system is visibly synchronous: The element puts out the sequence delayed by one clock period. In case of the division circuit an additional refinement injects a normalizing factor .1 so that we can use $0 \leq X \leq 1$ and $0 \leq Y \leq 1$ with the single restriction of $X \leq Y$

Again operation with negative numbers is possible but will not be discussed at this point.

An array computer using GCE's

By a slight extension of what was discussed in the previous section, we can define a computing element which performs on its two inputs a and b the operations a + b, a − b, a × b, as well as two types of division a÷ b (denoted by /) and b ÷ a (denoted by /). To this repertory we shall add the operation (A) → connection of output to a and (B) → connection of output to b.

It is then possible to design array computers of considerable power having a fixed structure. Figure 13 shows a three row, eleven column array with a "straight through" connection for the top row and a "down right" connection for the bottom rows. Programming this array now simply consists in defining the function of each GCE for the operation. An interesting example is furnished by functions F which can be expressed as sums of continuing fractions, i.e., $F = \Sigma\ f_i$ where f_i = continuing fraction. In this case there exists a simple 5-step algorithm to determine the operation to be performed by each element:

Step 1.: Using a "Polish Notation" in which operands precede operators and using the symbols + ,−, /, /, (A) and (B) defined above, write Σf_i as a "Polish String",PS, with (B) inserted after the first f_i.

Step 2.: Write each f_i as a PS with all products treated as single variables (called "product variables") and insert (B) after the first variable or "product variable" of the f_i.

Step 3.: Write each "product variable" of Step 2 in terms of individual variables in a PS. Insert (B) after each variable which is not a "product variable" and each first variable of a "product variable."

Step 4.: Combine Steps 1 through 3 into a single PS.

Step 5.: "Fill in" the functions of the elements by reading off the PS of Step 4. To this end start with the bottom left-hand element of the array and procede horizontally. When more than one operator occurs consecutively, fill in vertically until the next operand is reached. At the end, fill in all unspecified elements with the operator (A).

The figure shows as an example the set-up for the function

$$F = r/(s + t) + (gh)/\{(wt) + b\ [(dp) + y]\}$$

It is easily verified that the algorithm leads to the PS given below the expression for F and that the array effectively forms F.

A stochastic image transformer

As mentioned in the Introduction, one of the most attractive applications of stochastic computing elements is the simultaneous formation of very extensive linear combinations of variable values. We shall now describe a machine organization (which is as far removed from a Von Neumann machine as possible) which uses exclusively such linear combinations. Although we call this system an Image Transformer, it should be realized that "Signal Transformer" would be more appropriate. Figure 14 shows the schematic diagram of what is proposed: The object of the game is to map n^2 signals x_{ij} (i,j = 1, . . . , n) onto n^2 output signals y_{kl} (k, l = 1 , . . . , n) in such a fashion that each one of the y's is a weighted linear combination

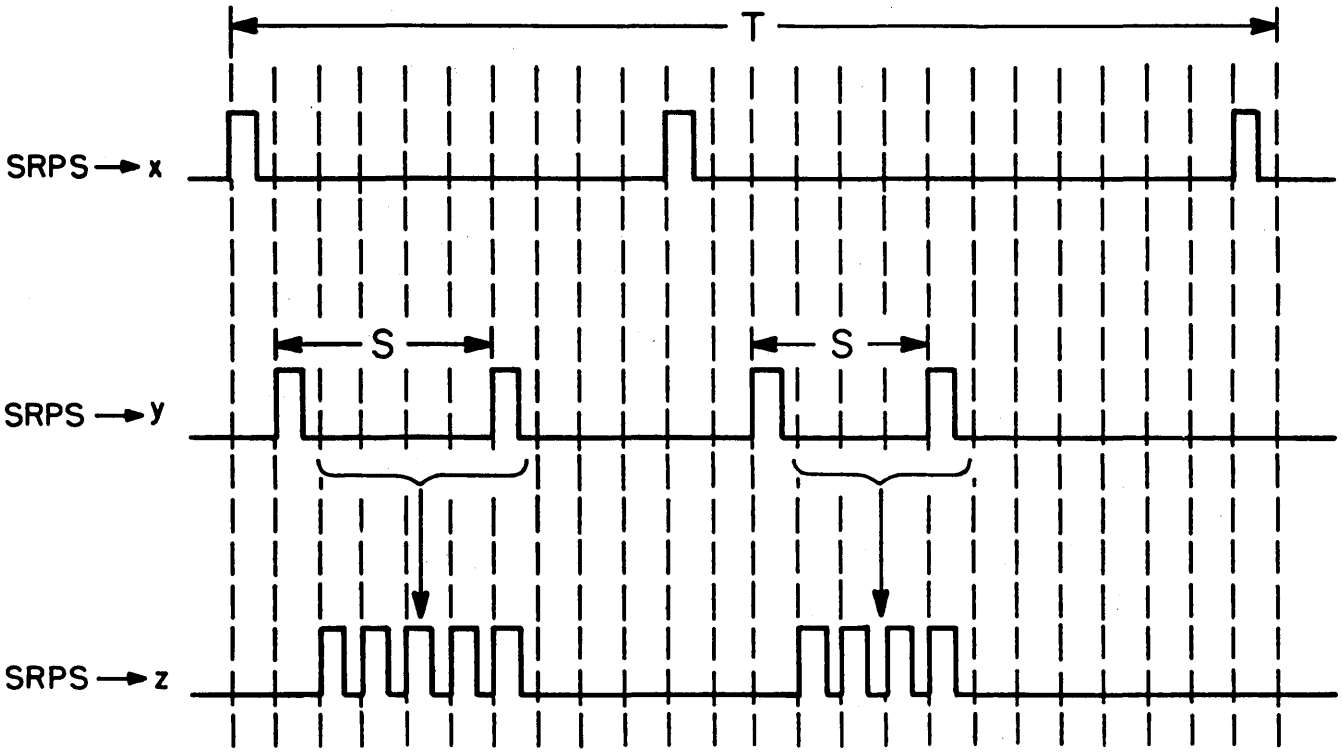
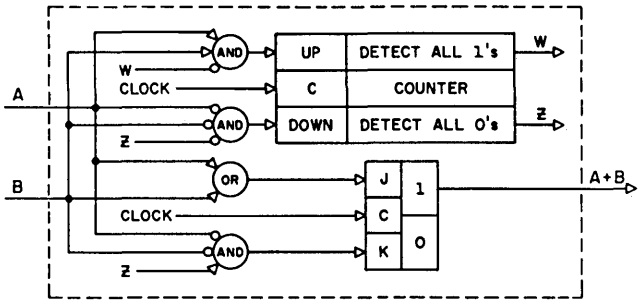


Figure 8 – Division of SRPS's



Fgure 9 – Addition of SRP's A and B

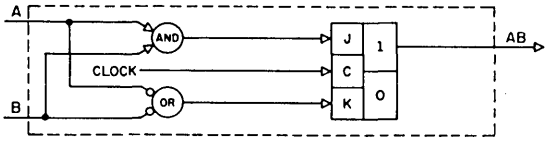


Figure 11 – Multiplication of SRPS's A and B

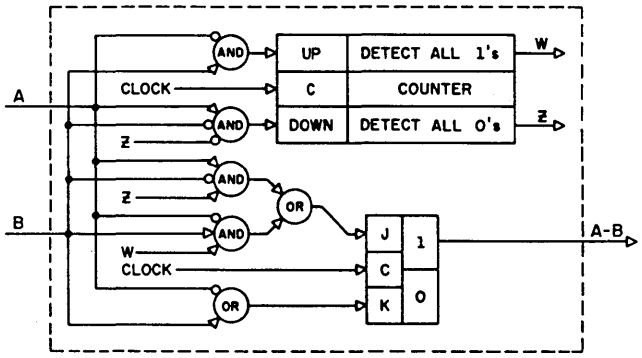


Figure 10 – Subtraction of SRPS B from SRPS A

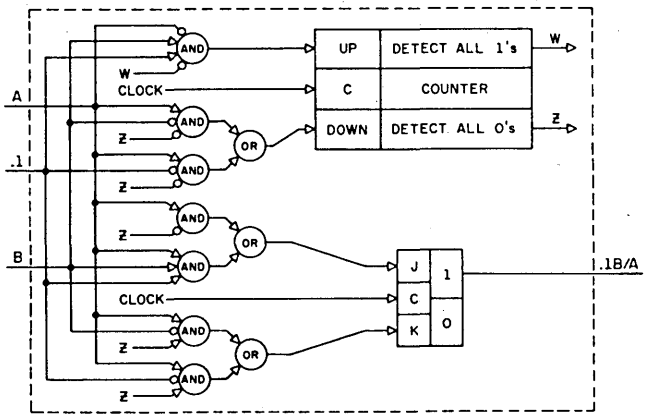


Figure 12 – Division of SRPS B by SRPS A and scaling

F = r/(s+t)+gh/(tw+b/(dp+y))

F = t(B)(B)s(B)+r(B)\(B)y(B)(B)p(B)dx+b(B)\t(B)wx+h(B)gx\+

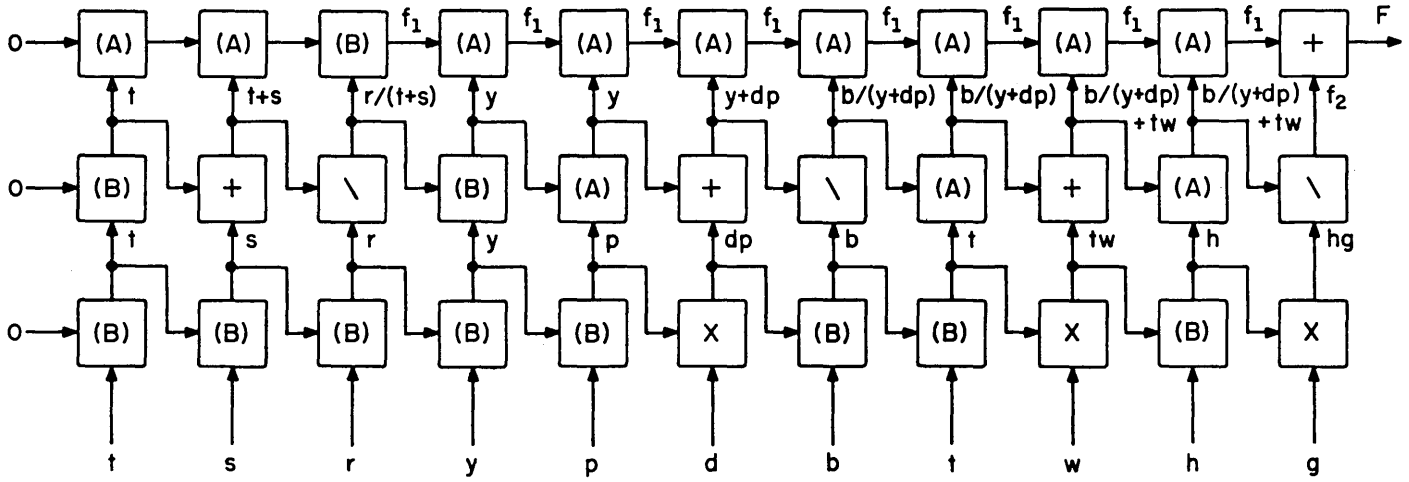


Figure 13 – Array of stochastic computing elements

of all x's. This corresponds to $y_{kl} = \sum_{ij} b_{klij} x_{ij}$. Evidently this mapping is not the most general possible but can be considered – after addition of a constant term a_{kl} – as the linear portion of the expansion of the most general function $f_{kl}(\ldots, x_{ij}, \ldots)$. It is then clear that the system produces an image mapping from input to output described by a general linear relationship and defined by the n^4 co-efficients b_{klij}. This general transformation encompasses:

1. Translations, rotations, and magnifications.
2. Conformal mappings.
3. Convolutions, and
4. Fourier Transforms.

By inspection it is easily seen that for a parallel organization in which each one of the n^2 output points collects its information through n^2 2-input AND's (each AND having x_{ij} and b_{klij} as inputs where k, l corresponds to the output point chosen), n^4 AND circuits will be necessary. Note that with respect to these the n^2 OR circuits can be neglected. The minimum number for n in a practical example would be 32: Even then over a million AND circuits would be required. This shows that a parallel organization is probably too costly. A way out is to consider a serial organization in which each of the output points is connected sequentially to n^2 AND circuits as described above. In this case higher circuit speeds are required since the on-line operation demands that all n^2 output points be scanned inside one flicker period, i.e., a 30th of a second. With circuit speeds of the order of 30 megacycles and demanding 10% accuracy (i.e., integration over 100 pulses) it is easily seen that the maximum n for the output matrix is about 200. This is not too far from television definition and it seems possible to reach a compromise in which the sequential system would be comparable to on-line video processors in both definition and speed. It should be noted, however, that the sequential scanning of the output necessitates a decoding matrix involving another n^2 (3-input) ANDs as shown in Figure 15, the inputs being the driving pulses for the two coordinate axes and the RPS coming from the AND combination described above. Note that there is perfect compatibility between the matrix selection pulses on the k and l lines and the stochastic signal carrying the information. It is now clear that the system will only require $2n^2$ ANDs and that the above mentioned figure of N = 200 leads to less than 100,000 AND circuits, a number not incompatible with cost considerations. The integrator at the output can be a tungsten light.

It is interesting to note that the determination of the b_{klij}'s hinges on the realization that i and j are (quantized) input coordinates and k and l (quantized) output coordinates. Assuming i, j, k and l continuous, we would therefore write a general transformation as

$$\left.\begin{aligned} k &= k\ (i,J) \\ l &= l\ (i,j) \end{aligned}\right\} \qquad (6.1)$$

The identity transformation k = i and l = j gives, obviously, rise to

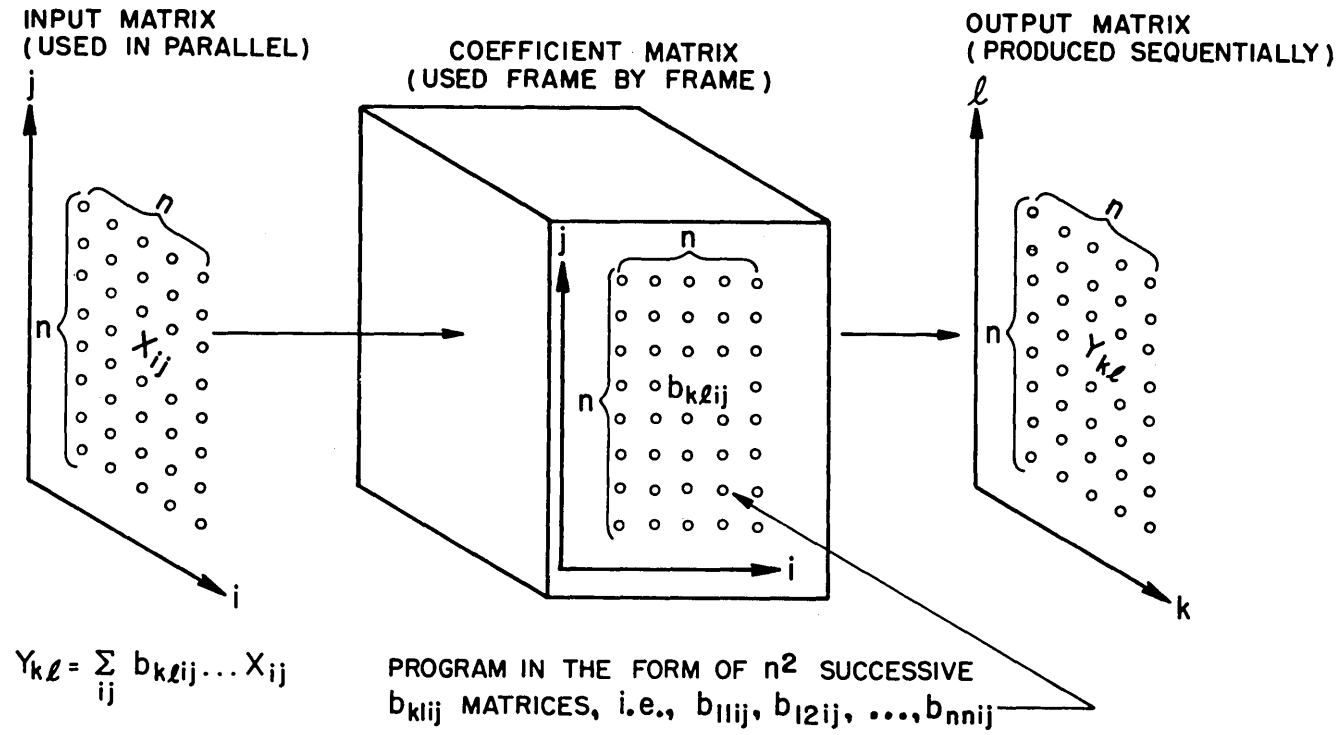


Figure 14 – Transformatrix

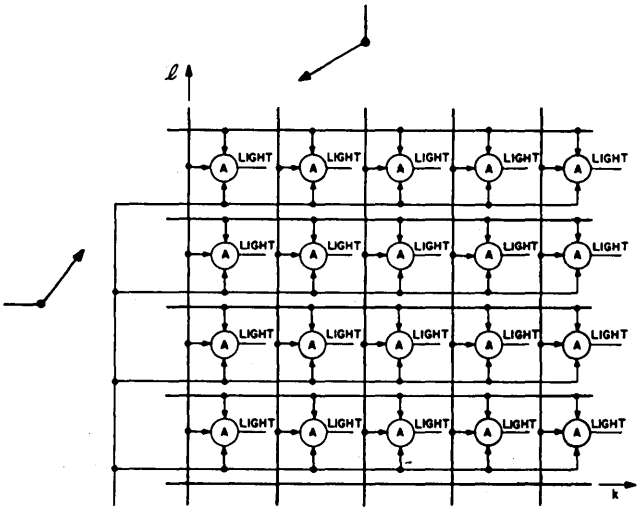


Figure 15 – Scanning matrix for the stochastic image transformer

$$b_{klij} = \delta(k - i)\,\delta(l - j) \qquad (6.2)$$

Similarly it is easy to see that a Fourier Transformation corresponds to

$$y_{kl} = \sum_i \sum_j x_{ij} e^{(ki+lj)v\overline{-1}} \qquad (6.3)$$

The real and imaginary parts of $e^{(ki+lj)}$ give the b_{klij}'s immediately: The latter will have real and imaginary parts. Short of displaying $(y_{kl})^2$, it might therefore be necessary to split the output matrix into two parts.

REFERENCES

1 S T RIBEIRO
Comments on pulsed-data hybrid computers
IEEE Transactions on Electronic Computers Vol EC-13 October 1964

2 L O GILSTRAP H J COOK C W ARMSTRONG
Study of large neuromime networks
Adaptronics Interim Engineering Report No 1 to the Air Force Avionics Laboratory USAF Wright-Patterson AFB Ohio August 1966

3 C AFUSO
Quarterly technical progress reports
Circuit Research Section Department of Computer Science University of Illinois starting January 1965 to present

4 B R GAINES
Techniques of identification with the stochastic computer
IFAC Symposium 1967 Prague

5 B R GAINES
Stochastic computer thrives on noise
Electronics Vol 40 No 14 July 10 1967 p 72-79

6 S T RIBEIRO
Random-pulse machines
IEEE Transactions on Electronic Computers Vol EC-16 June 1967

A practical method for comparing numerical integration techniques

by PAUL P. SHIPLEY
Space Division
North American Rockwell Corporation
Downey, California

INTRODUCTION

Review of basic state variable notation

Most schemes for numerical integration of a set of differential equations are closely related to a corresponding analog simulation. The equations to be solved are reduced to a set of first order differential equations, each of which requires a single integration.[1] Hence, one way to set up the problem is to draw an analog signal flow graph, then write an equation for the input to each integrator. Simultaneous numerical integration of these equations is then performed.

In an analog computer, the total input to each integrator is a function of other integrator outputs, inputs from external sources and time. Thus, if there are m integrator outputs x_i ($i = 1, 2, \ldots, m$), and n inputs from external sources u_i ($i = 1, 2, \ldots, n$), the total input to the ith integrator may be expressed by equation (1).

$$x_i = f_i\,(x_1, x_2, \ldots, x_m; u_1, u_2, \ldots, u_n, t). \quad (1)$$

The entire system may be represented by a set of equations in the form of equation (1), one for each integrator. These equations may be combined in matrix form as in equation (2).

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_m \end{bmatrix} = \begin{bmatrix} f_1\,(x_1, x_2, \ldots, x_m; u_1, u_2, \ldots, u_n, t) \\ f_2\,(x_1, x_2, \ldots, x_m; u_1, u_2, \ldots, u_n, t) \\ \\ f_m\,(x_1, x_2, \ldots, x_m; u_1, u_2, \ldots, u_n, t) \end{bmatrix} \quad (2)$$

This is commonly written in a more compact form as

$$\dot{x} = f\,(x, u, t) \quad (3)$$

where a bar under a letter denotes a column matrix as follows:

$$\underline{x} \triangleq \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad \underline{u} \triangleq \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}, \quad \underline{f} \triangleq \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} \quad (4)$$

Figure 1 is the analog of equation (3).

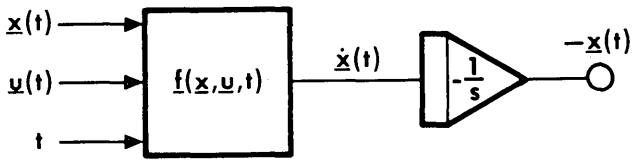


Figure 1—Analog of state equations

The elements x_i of x are commonly referred to as state variables, and together are sufficient to describe the state of the system. The state of the system at any time, t , may be determined from the state at some previous time, t_o , provided all n elements of the input matrix u are known for all time between t_o and t . This is accomplished simply by allowing the analog computer to operate normally. The elements x_i of the matrix x are commonly associated with components of a vector in m-dimensional space. Hence, x is commonly referred to as the state vector. Similarly, u is the control vector.

Stability analysis using Lyapunov's Second Method

In applying Lyapunov's Second Method,[2] the practical objective is to show that a positive definite function of the state variables becomes smaller as time progresses. A positive definite function is one which is

always positive unless all of the state variables are zero, in which case the function is zero. Such a function may be considered a generalized energy function and is sometimes termed a Lyapunov function. Stability of a system is demonstrated by deriving a suitable Lyapunov function, then showing that the time rate of change of that function is always negative, or at least never positive.

At the present time, the most serious shortcoming of Lyapunov's Second Method is that no general approach is known for deriving suitable Lyapunov functions. This limits application to a few special cases. Nevertheless, it is a practical and powerful method for comparing numerical integration schemes for simulation of linear and nonlinear systems.

Derivation of specific Lyapunov functions is not necessary because only a relative measure of stablity is required. That is, the absolute stability of a simulation is not nearly as important as how the simulation compares with the actual system, or how various numerical integration algorithms compare with each other.

So far, the main practical application has been to a nonlinear parameter tracking computer which was subjected to several input signals and which sometimes included time-varying gains. However, for the present discussion, an autonomous (i.e., no outside inputs and time invariant) nonlinear system will be assumed, as in conventional Lyapunov theory. Under these conditions, equation (3) is reduced to

$$\underline{\dot{x}} = \underline{f}\,(\underline{x}) \tag{5}$$

Assume that a Lyapunov function V(x)exists. By the chain rule of differentiation, the time rate of change of V (x) is given by

$$\dot{V}(\underline{x}) = \sum \frac{\partial V}{\partial x_j}\,\dot{x}_j = \sum \frac{\partial V}{\partial x_j}\,f_j\,(\underline{x}) = V'_{\underline{x}}\,\underline{f}\,(\underline{x}) \tag{6}$$

where the gradient V_x is defined by equation

(7)

$$V' = \left(\frac{\partial V}{\partial x_1}, \frac{\partial V}{\partial x_2}, \ldots, \frac{\partial V}{\partial x_m}\right) \tag{7}$$

and the symbol (′) denotes a transposed matrix.

For simulation purposes, one may assume a quadratic Lyapunov function. Although it is possible to obtain some results in terms of a more general function, it is unnecessary to choose a function which precisely encompasses the system stability boundary,[3] since only a relative measure of stability is required. Hence, the function V (x) need not always be defined rigorously as a Lyapunov function to provide a good measure of deterioration in system performance due to sampling and numerical integration. A quadratic Lyapunov function is of the form

$$V = \underline{x}'\,A\,\underline{x} = \sum_i \sum_j a_{ij}\,x_i\,x_j \tag{8}$$

where A denotes a square, symmetric, positive definite matrix. For example, if A is a unit matrix, V is the sum of the squares of the state variables (integrator outputs). Equation (6) then becomes

$$\dot{V}\,(\underline{x}) = 2\underline{x}'\,A\,\underline{\dot{x}} = 2\underline{x}'\,A\,\underline{f}\,(\underline{x}) \triangleq g\,(\underline{x}). \tag{9}$$

Equation (9) will serve as a standard against which similar expressions derived for numerical integration algorithms will be compared.

Application to specific integration algorithms

For numerical analysis, the set of first order differential equations represented by equation (5) is approximated by a set of difference equations. For rectangular integration,

$$\underline{x}_{k+1} = \underline{x}_k + T\,\underline{f}\,(\underline{x}_k) \tag{10}$$

where $\underline{x}_k$ denotes the value of $\underline{x}(t)$ at the k'th sampling instant and T denotes the sampling period. At the k'th instant, equation (8) becomes

$$V_k = \underline{x}'_k\,A\,\underline{x}_k \tag{11}$$

Taking the first difference of equation (11) with respect to k yields an expression analogous to equation (9):

$$V_{k+1} - V_k = \underline{x}'_{k+1} A \underline{x}_{k+1} - \underline{x}'_k\,A\underline{x}_k = (\underline{x}_{k+1} + \underline{x}_k)'\,A(x_{k+1} - x_k) \tag{12}$$

The simulation is stable provided this difference is always negative. If equation (10) is substituted into equation (12), and equation (9) into the result, one obtains

$$V_{k+1} - V_k = (2\underline{x}_k + T\,\underline{f}\,(\underline{x}_k))'\,AT\,\underline{f}\,(\underline{x}_k) = T\,[g\,(\underline{x}_k) + T\,f'\,(\underline{x}_k)\,A\,f\,(\underline{x}_k)]. \tag{13}$$

The first term on the right of equation (13) corresponds directly to the term on the right of equation (9) for the actual system. The second term is always positive for a positive definite matrix A, and therefore indicates a reduction in stability due to sampling.

In general, all that is required to compare stability of various numerical integration schemes is to derive an expression similar to equation (13) for each, then compare the terms contributed by sampling.

If the input to a rectangular integrator is advanced by one sample (not always feasible in closed-loop systems), the equivalent of equation (10) is

$$\underline{x}_{k+1} = \underline{x}_k + T\,\underline{f}\,(\underline{x}_{k+1}). \tag{14}$$

Substituting equation (14) into equation (12) gives

$$V_{k+1} - V_k = T\,[g\,(\underline{x}_{k+1}) - T\,\underline{f}'\,(\underline{x}_{k+1})\,A\,f\,(\underline{x}_{k+1})]. \quad (15)$$

Comparison of equations (9), (13), and (15) reveals that advancing the input by one sample makes the simulation more stable than the actual system, leading to overly optimistic answers. On the other hand, if the input is not advanced, the computational problem is simplified, but the results will be pessimistic.

If trapezoidal integration is used (again not always feasible in closed-loop systems),

$$\underline{x}_{k+1} = \underline{x}_k + \frac{T}{2}\,[\underline{f}\,(\underline{x}_{k+1}) + \underline{f}\,(\underline{x}_k)]. \quad (16)$$

Substituting equation (16) into equation (12) gives

$$V_{k+1} - V_k = \frac{T}{2}\,[\underline{x}_{k+1} + \underline{x}_k]'A[\underline{f}\,(\underline{x}_{k+1}) + \underline{f}\,(\underline{x}_k)]. \quad (17)$$

For a linear system

$$f\,(x_{k+1}) + f\,(x_k) = f\,(x_{k+1} + x_k). \quad (18)$$

Substituting equations (18) and (19) into (17) gives

$$V_{k+1} - V_k = \frac{T}{4}\,g\,(\underline{x}_{k+1} + \underline{x}_k). \quad (19)$$

Hence, it is apparent that the stability boundary of a simulation based upon trapezoidal integration (Tustin's rule) exactly matches that of the continuous system for the linear case, as is known from z-transform theory;[4] however, this is not strictly true for the nonlinear case.

It would seem to be desirable to have a simulation technique capable of accurately locating the stability boundary for a broad class of nonlinear systems. This may be accomplished by using an algorithm which reduces to trapezoidal integration for linear systems, but which is equally valid for nonlinear systems:

$$\underline{x}_{+1} = \underline{x}_k + \frac{T}{2}\,\underline{f}\,[\underline{x}_{k+1} + \underline{x}_k] \quad (20)$$

Substituting equation (20) into equation (12) and equation (9) into the result gives equation (19) directly without assuming linearity.

These results have been shown only for areas of phase space where a quadratic Lyapunov function is suitable. In general, however, the same is true for all linear autonomous systems and for some important non-linear systems. In any case, a direct measure of errors due to sampling and numerical integration is obtained for all types of systems.

A practical example

The basic technique presented here has proven of great value in evaluating numerical integration schemes for a parameter tracking scheme for an adaptive flight control system for the X-20 gliding reentry vehicle simulator.[5] In this case, exact answers are readily obtained, and it is the only known method which will give results with the required precision.

The basic problem is the computation of coefficients of a linear differential equation from measured data. Such an equation has the form

$$x_o + \Sigma b_j x_j = 0 \quad (21)$$

where the b_j's are the coefficients to be determined, and the x_j's are various measurable functions and their derivatives.

During parameter adjustment, the current estimate of the i'th parameter b_i will be designated by b^*_i. Estimation error is defined by equation (22).

$$e = x_o + \Sigma b^*_j x_j \quad (22)$$

Subtracting equation (21) from equation (22) yields

$$e = \Sigma\,(b^*_j - b_j)\,x_j \quad (23)$$

Automatic coefficient adjustment is achieved with a gradient technique[6] to reduce squared error. The adjustments are made in accordance with equation (24)

$$\dot{b}^*_j = -K\partial e^2/\partial b^*_j = -2K\,e\,x_j. \quad (24)$$

where K is any positive constant.

It has been shown[7] that stability of the adjustments may be verified using Lyapunov's Second Method. A Lyapunov function V is defined in terms of deviations of the estimated parameters from the true parameters by equation (25).

$$V \triangleq \Sigma\,(b^*_j - b_j)^2 \quad (25)$$

If the rate of change of V with respect to time can be made negative or zero at all times, the adjustments are said to be stable in the sense of Lyapunov. Differentiating equation (25) with respect to time, and assuming the true system parameters b_j are constant yields

$$\dot{V} = 2\,\Sigma\,(b^*_j - b_j)\,\dot{b}^*_j. \quad (26)$$

Substituting equation (24) into equation (26) and (23) into the result yields

$$\dot{V} = -4K\,e\,\Sigma\,(b^*_j - b_j)\,x_j = -4K\,e^2. \quad (27)$$

Whenever any error occurs, the parameters adjust to reduce the magnitude of the Lyapunov function. The adjustment loops are therefore stable. An adaptive control system for the X-15 vehicle using a refined version of this parameter tracking scheme was simulated on an analog computer.[8,9] Results were accurate and rapid convergence was achieved.

Further work on the X-20 simulator employed a digital computer to perform the parameter tracking function.[5] This required some form of numerical integration. The analytical method for evaluating numerical integration techniques was used as follows to choose an integration formula:

For the discrete case, equation (25) becomes

$$V_k \triangleq \Sigma (b^*_{jk} - b_j)^2 \qquad (28)$$

where V_k is the value of V at the k'th instant, and b^*_{jk} is the value of the j'th parameter at the k'th instant. Taking the first difference of equation (28) with respect to k gives

$$V_{(k+1)} - V_k = \Sigma[(b^*_{jk+1} - b_j)^2 - (b^*_{jk} - b_j)^2]$$
$$= \Sigma (b^*_{j(k+1)} - b^*_{jk})\,(b^*_{j(k+1)} + b^*_{jk} - 2b_j). \qquad (29)$$

If this first difference is always negative, the adjustments will always be stable. Several cases will now be considered.

If equation (24) is evaluated by rectangular integration based upon past input,

$$b^*_{j(k+1)} = b^*_{jk} - 2KT\, e_k\, x_{jk} \qquad (30)$$

Substituting equations (30) and (23) into equation (29) yields

$$V_{k+1} - V_k = -4KT\, e_k{}^2[1 - KT \Sigma x_{jk}{}^2] \qquad (31)$$

Comparison with equation (27) reveals that the second term in brackets is due to sampling and is destabilizing. Equation (31) corresponds directly to equation (13) for the more general case discussed earlier.

For rectangular integration based upon present input,

$$b^*_{j(k+1)} = b^*_{jk} - 2KT\, e_{k+1}\, x_{j(k+1)} \qquad (32)$$

and

$$V_{k+1} - V_k = -4KT\, e_{k+1}{}^2[1 + KT \Sigma x_{j(k+1)}{}^2]. \qquad (33)$$

It is not feasible to implement this approach due to the difficulty involved in computing e_{k+1} and $x_{j(k+1)}$. If it could be implemented, the parameter adjustment loops would always be stable, even more so than the continuous system.

For trapezoidal integration,

$$b^*_{j(k+1)} = b^*_{jk} - KT\,(e_{k+1}\, x_{j(k+1)} + e_k x_{jk}) \qquad (34)$$

and

$$V_{k+1} - V_k = -2KT\,[e^2_{k+1} + e^2_k + \frac{KT}{2}$$
$$(e^2_{k+1} \Sigma x^2_{j(k+1)} - e^2_k \Sigma x^2_{jk})]. \qquad (35)$$

For relatively high sampling rates, the higher order terms almost cancel, and stability is nearly the same as for the continuous system. Equation (30) is inferior to equations (32) and (34), but is sometimes the only one which can be implemented physically. An attempt to circumvent implementation problems was made by using linear input prediction for each integrator. Combining the predicted inputs with equations (32) and (34) leads to open-ended rectangular and trapezoidal formulae. Heun's method, consisting of a single pass through a rectangular predictor and trapezoidal corrector was also analyzed.

For the application cited, ordinary rectangular integration as described by equation (30) turned out to be superior to all others considered. One reason for this result was that the more complex formulae introduced higher-order response modes into the parameter adjustments, and were therefore not as stable as one might hope, even though more computing time was required. Of course, this result does not mean that rectangular integration would be superior for all numerical integration applications; but, definitive analytical results were obtained for this particular example. Additional results were also obtained for non-steepest descent (i.e., unequal integrator gains) and for time-varying integrator gains.

Not only was it possible to select a numerical integration scheme but it was also possible to find the maximum stable gain and to optimize gain for maximum convergence rate. The maximum stable gain was obtained by observing from equation (31), that the adjustment is stable only if the term in brackets is positive or zero. The maximum stable gain is therefore

$$K_{max} = 1/(T \Sigma x^2_{jk}) \qquad (36)$$

Equation (36) also gives the maximum permissible sampling period T for a given gain K. The optimum gain is found by differentiating equation (31) with respect to K and setting the result equal to zero.

$$K_{opt} = 1/(2T \Sigma x^2_{jk}) = K_{max}/2. \qquad (37)$$

Substituting equation (37) into (30), the optimum adjustment algorithm is

$$b^*_{j(k+1)} = b^*_{jk} - e_k x_{jk}/\Sigma x^2_{ik}. \qquad (38)$$

Relative stability

Now that an approach suitable for stability analysis of nonlinear systems has been developed, extensions to concepts other than stability should be considered. As an example, it has been shown that the stability boundary of a simulation based upon Tustin's rule (trapezoidal integration operator) exactly duplicates that of any linear system being simulated. However, it is readily shown by stability charts obtained through mapping techniques[10] that damping and frequency of poles not on the stability boundary are changed appreciably in some instances. It would seem, then, that a complete evaluation of numerical integration schemes would involve some evaluation of relative stability and possibly frequency, as well as absolute stability.

Normally, the Lyapunov theory only provides a measure of absolute stability, much the same as the Routh-Hurwitz criterion for linear systems. A well-known technique for extending the Routh-Hurwitz criterion to obtain a measure of relative stability is the translation of all s-plane poles in the positive real direction until instability occurs. The amount of translation required is a measure of relative stability. Thus, the Routh-Hurwitz criterion can be used to derive much the same type of information as the root loci method or frequency response plots. The transformation requires a substitution of (s—a) for s in the transfer function. This translates all roots (a) units to the right, as illustrated in Figure 2. This basic technique may actually be used to factor the characteristic equation by identifying the location of every s-plane pole.

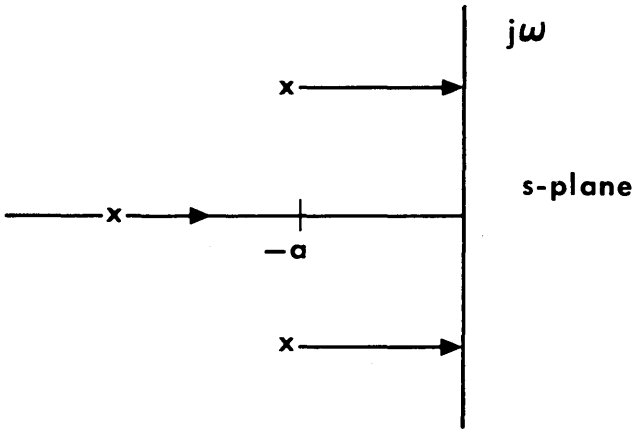


Figure 2—Translation of s-plane poles

The transformation may also be carried out in the time domain by multiplying all variables by e^{at} as indicated by the following Laplace transform theorem:

$$L[e^{-at} f(t)] = F(s+a) \tag{39}$$

An important advantage of the time domain transformation is that it may also be applied to nonlinear systems, where frequency domain concepts are not too meaningful.

As an example, consider the nonlinear system described by equation (5), repeated here as equation (40).

$$\dot{\underline{x}} = \underline{f}(\underline{x}) \tag{40}$$

and a quadratic Lyapunov function

$$V = \underline{x}' A \underline{x}. \tag{41}$$

Differentiating equation (41),

$$\dot{V} = 2\underline{x}' A \underline{\dot{x}} = 2\underline{x}' A \underline{f}(x) \triangleq \underline{g}(\underline{x}). \tag{42}$$

A new transformed set of variables which is less stable than the original set is defined by

$$\hat{\underline{x}} \triangleq e^{at} \underline{x}. \tag{43}$$

Differentiating equation (43), and substituting (40),

$$\dot{\hat{\underline{x}}} = e^{at}(\dot{\underline{x}} + a\underline{x}) = e^{at}(\underline{f}(\underline{x}) + a\underline{x}) \tag{44}$$

The Lyapunov function for the transformed system is

$$V = \underline{x}' A \underline{x} \tag{45}$$

Differentiating equation (45),

$$\dot{V} = 2\dot{\hat{\underline{x}}}' A \hat{\underline{x}} = 2e^{2at} \underline{x}' A (\underline{f}(\underline{x}) + a\underline{x}) \tag{46}$$
$$= e^{2at}(g(\underline{x}) + 2a\, \underline{x}' A \underline{x}).$$

Comparison with equation (42) clearly reveals the terms in equation (46) which are a consequence of the destabilizing transformation. Equation (46) can now be used as a standard against which various numerical integration schemes may be compared at various levels of stability. If it is desired to study the effects of sampling upon the frequency characteristics of the response, the coefficient (a) may be chosen to be complex, at least for linear systems, and possibly for nonlinear limit cycles.

For the sampled-data case, the equivalent of equation (43) is

$$\hat{\underline{x}}_k = e^{akT} \underline{x}_k \tag{47}$$

and the corresponding Lyapunov function is

$$\hat{V}_k = \hat{\underline{x}}'_k A \hat{\underline{x}}_k. \tag{48}$$

Taking the first difference of equation (48) with respect to k and substituting equation (47),

$$\hat{V}_{k+1} - \hat{V}_k = \hat{x}'_{k+1} A \hat{\underline{x}}_{k+1} - \hat{\underline{x}}'_k A \hat{\underline{x}}_k \tag{49}$$
$$= e^{2a(k+1)T}(\underline{x}'_{k+1} A \underline{x}_{k+1} - e^{-2aT} \underline{x}'_k A \underline{x}^k).$$

At this point, various numerical integration formulae may be compared by substitution into equation (49). For example, rectangular integration may be studied by substituting equation (10). Then,

$$\hat{V}_{k+1} - \hat{V}_k = Te^{2a(k+1)T}[g(\underline{x}_k) + (1e^{-2aT}) \tag{50}$$
$$\underline{x}'_k A \underline{x}_k / T + T \underline{f}'(\underline{x}_k) A f(\underline{x}_k)].$$

Comparing equation (50) with equations (42), (46), and (13) reveals the following:

(1) The first term within the brackets of equation (50) is associated with stability of the actual system being simulated.
(2) The second term and leading coefficient is associated with the transformation of equation (47).
(3) The final term is the degradation in stability associated with sampling and rectangular integration.

The ability to isolate terms due to various types of deterioration in performance, then compare them for various types of numerical integration is one of the primary advantages of the method.

Analysis of noise and roundoff error

Noise added to integrator inputs in a simulation produces separate terms in the Lyapunov function. Hence, the Lyapunov method is suitable for evaluating the sensitivity of various numerical integration algorithms to this type of disturbance. Roundoff error is essentially equivalent to a random noise, and will be treated as such in this analysis. If noise is added to the integrator inputs, the state vector equation for the continuous system is of the form:

$$\dot{\underline{x}} = \underline{f}(\underline{x}) + \underline{n} \tag{51}$$

Assuming the same Lyapunov function as in equation (8),

$$\dot{V} = 2\underline{x}' A\dot{\underline{x}} = 2\underline{x}'A[f(x)+\underline{n}]=g(\underline{x}) + 2\,\underline{x}'\,\underline{An} \tag{52}$$

The first term on the right side of the equation (52) is associated with the stability of the continuous noise free system, while the final term is a consequence of the noise. For rectangular integration,

$$\underline{x}_{k+1} = \underline{x}_k + T\underline{f}(\underline{x}_k) + T\underline{n}_k \tag{53}$$

Substituting into equation (12),

$$\begin{aligned} V_{k+1} - V_k &= (\underline{x}_{k+1} + \underline{x}_k)'A(\underline{x}_{k+1} - \underline{x}_k) \\ &= [2x_k + T\underline{f}(\underline{x}_k) + T\underline{n}_k]'A[Tf(\underline{x}_k) + T\underline{n}_k] \\ &= Tg(\underline{x}_k) + 2\underline{x}_k' AT\underline{n}_k + T^2[\underline{f}'(\underline{x}_k)Af(\underline{x}_k) \\ &\quad + \underline{n}'_k A[\underline{n}_k + 2f(\underline{x}_k)]]. \end{aligned} \tag{54}$$

The first two terms in equation (54) correspond to terms in equation (52) for the continuous system and the third term indicates destabilization due to rectangular integration. The final term represents interaction between noise and the numerical integration scheme.

From a statistical point of view, roundoff error is essentially independent of the variable being rounded, provided the error is small. Taking the expected value of equation (54) under these assumptions,

$$\begin{aligned} E[V_{k+1} - V_k] = E[&Tg(\underline{x}_k) + T^2f'(\underline{x}_k)\,Af(\underline{x}_k) \\ &+ T^2n'_k\,An_k]. \end{aligned} \tag{55}$$

The only long-term effect of quantization error appears in the final term of equation (55), which is positive definite, and must, therefore, tend to increase the Lyapunov function. By comparing this term with similar terms generated by other numerical integration algorithms, a comparative evaluation may be made.

For trapezoidal integration with noise,

$$\underline{x}_{k+1} = \underline{x}_k + \frac{T}{2}[\underline{f}(\underline{x}_{k+1}) + \underline{f}(\underline{x}_k)] + T\underline{n}_k. \tag{56}$$

Then,

$$\begin{aligned} V_{k+1} - V_k &= (\underline{x}_{k+1} + \underline{x}_k)'A(\underline{x}_{k+1} - \underline{x}_k) \\ &= (\underline{x}_{k+1} + \underline{x}_k)'A\{\frac{T}{2}[f(\underline{x}_{k+1}) + f(\underline{x}_k)] + T\underline{n}_k\}' \\ &= \frac{T}{2}[\underline{x}_{k+1} + \underline{x}_k]'\,A[\underline{f}(\underline{x}_{k+1}) + f(\underline{x}_k)] + 2T\underline{x}_k'\,A\underline{n}_k \\ &\quad + Tn'_k\,A\{Tn_k + \frac{T}{2}[f(x_{k+1}) + f(x_k)]\}. \end{aligned} \tag{57}$$

The final terms in equations (54) and (57) agree quite closely for small values of T, indicating that propagation of integrator noise and roundoff error is similar in the two cases.

Predictor-corrector methods

Application to predictor-corrector methods of numerical integration is of interest in view of the great popularity of such methods. A simple example is a single iteration using a rectangular predictor and a trapezoidal corrector.

$$\underline{x}^p{}_{k+1} = \underline{x}_k + T\underline{f}(\underline{x}_k) \tag{58}$$

$$\underline{x}_{k+1} = \underline{x}_k + \frac{T}{2}[\underline{f}(\underline{x}^p{}_{k+1} + \underline{f}(\underline{x}_k)]. \tag{59}$$

Then,

$$\begin{aligned} V_{k+1} - V_k &= (\underline{x}_{k+1} + \underline{x}_k)'A(\underline{x}_{k+1} - \underline{x}_k) \\ &= \{2\underline{x}_k + \frac{T}{2}[f(\underline{x}^p{}_{k+1}) + f(\underline{x}_k]\}'A\,\frac{T}{2}[\underline{f}(\underline{x}^p{}_{k+1}) \\ &\quad + f(\underline{x}_k)] = T\,\underline{x}'_k A[\underline{f}(\underline{x}^p{}_{k+1}) + \underline{f}(\underline{x}_k)] + \frac{T^2}{4} \\ &\quad [f(\underline{x}^p{}_{k+1}) + \underline{f}(\underline{x}_k)]'A[\underline{f}(\underline{x}^p{}_{k+1} + \underline{f}(\underline{x}_k)]. \end{aligned} \tag{60}$$

In the example,

$$\begin{aligned} V_{k+1} - V_k = -T(&e^p{}_{k+1}\,e_{k+1} + e_k^2 + KT \\ &(e^{p2}{}_{k+1}\,\Sigma\,x^2{}_{j(k+1)} - e^2_k\,\Sigma\,x^2{}_{jk})) \end{aligned} \tag{61}$$

where

$$\begin{aligned} e^p{}_{k+1} = (e_{k+1} - KT\,e_k\,\Sigma\,x_{ik}\,x_{i(k+1)})/ \\ (1\text{-}KT\,\Sigma\,x^2{}_{i(k+1)}) \end{aligned} \tag{62}$$

When equation (62) is substituted into equation (61), the denominator of equation (62) appears as the denominator of equation (61). Hence, stability is assured only if equation (36) is satisfied, which is identical to the corresponding condition without the trapezoidal corrector. This is in complete agreement with linear theory for integration of a first order system, which shows that stability is not improved no matter how many times the corrector is repeated.[4] Results may also be obtained for the case where the corrector is iterated several times during each sampling period.

Higher order methods

Higher order integration alogrithms have been analyzed to some extent, but no real application has been found; for this reason, no detailed treatment will be given, but the approach will be illustrated briefly with Hamming's Corrector:"

$$y_{n+1} = 1/8[9y_n - y_{n-2}+3T(x_{n+1} + 2x_n - x_{n-1})] \quad (63)$$

or in matrix form,

$$\underline{y}_{n+1} = B\, y_n + T\, f_{n+1} \quad (64)$$

where

$$\underline{y}'_{n+1} = (y_{n+1}\ y_n,\ y_{n-1}) \quad (65a)$$

$$B = \begin{bmatrix} 9/8 & 0 & -1/8 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (65b)$$

$$f'_{n+1} = (3(x_{n+1}+2x_n - x_{n-1})/8,\ 0,\ 0). \quad (65c)$$

and the Lyapunov function for a first order system is of the form

$$V_{n+1} = y'_{n+1}\, Ay_{n+1} \quad (66)$$

Then

$$V_{n+1} - V_n = y'_{n+1}\, Ay_{n+1} - y'_n\, Ay_n \quad (67)$$

$$= y'_n\,(B'AB-A)\ y_n + 2T\ y_n\ 'B'A\ f_{n+1} + T^2\ f'_{n+1}\ Af_{n+1}$$

It is convenient to set a_{22} equal to unity, expand B'AB-A, and set the off diagonal terms equal to zero. This gives the following positive definite matrix for A:

$$A = \begin{bmatrix} 64 & -8 & -8 \\ -8 & 1 & 1 \\ -8 & 1 & 1 \end{bmatrix} \quad (68)$$

From this point on, the analysis is fairly straightforward.

CONCLUSIONS

A technique has been presented for analysis and comparison of numerical integration methods for both linear and nonlinear systems. Results apply to quadrature formulae, as well as predictor-correctors, although applications to date have used only simple formulae. In general, this approach is useful in some real-time simulations, or in implementing digital control systems with computers of limited capability. For more general simulation work, where computer capability and computation time are not critical, Runge-Kutta integration[11] is excellent, combining the high accuracy of predictor-correctors with the stability and freedom from extraneous response modes of simple quadrature formulae and operational methods (such as the z-form).[4]

ACKNOWLEDGMENTS

The research reported in this paper was supported in part by the Air Force Flight Dynamics Laboratory, Research and Technology Division, Wright-Patterson Air Force Base, Ohio, under contract AF 33(615)-2459.

REFERENCES

1 J T TOU
Modern control theory
McGraw-Hill Book Company Inc New York chap 3 1964

2 J E GIBSON
Nonlinear automatic control
McGraw-Hill Book Company Inc New York chap 8 1963

3 W HAHN
Theory and application of Lyapunov's direct method
Prentice-Hall Inc Englewood Cliffs NJ chap 4 p 80 1963

4 J W HUNG et al
Investigation of numerical techniques as applied to digital flight control systems
AFFDL-TR-66-68 North American Aviation Inc Air Force Flight Dynamics Laboratory Wright-Patterson Air Force Base Ohio October 1966

5 P P SHIPLEY et al
AFFDL-TR-66-32 CONFIDENTIAL
North American Aviation Inc Air Force Flight Dynamics Laboratory Wright-Patterson Air Force Base Ohio January 1966

6 T F POTTS et al
The automatic determination of human and other system parameters
Proc Western Joint Computer Conference Los Angeles California pp 645-660 May 1961

7 B J MILLER
A theoretical and analog study of a steep descent coefficient computer for process analysis and adaptive control
PhD Dissertation Ohio State University 1962

8 P P SHIPLEY
An adaptive flight control system
ASME Winter Annual Meeting and Energy Systems Exposition New York November 30 1966

9 P P SHIPLEY et al
Self-adaptive flight control by multivariable parameter identification
AFFDL-TR-65-90 North American Aviation Inc Air Force Flight Dynamics Laboratory Wright-Patterson Air Force Base Ohio May 1965

10 *Final report investigation and evaluation of DDOFTT mathematical procedures*
MSR 62-08 the Moore School of Electrical Engineering University of Pennsylvania Philadelphia 1962

11 A RALSTON
A first course in numerical analysis
McGraw-Hill Book Company Inc New York chap 5 1965

AutoSACE–automatic checkout for Poseidon

by G. W. SCHULTZ
and J. M. COLEBANK
Lockheed Missiles & Space Company
Sunnyvale, California

INTRODUCTION

A major difficulty encountered in the manufacture of complex systems is the insurance of specification compliance of sub-assemblies at numerous points in the manufacturing process. An additional problem is encountered when historical data concerning any particular specification failure is required.

Both of these problems arise in the development and construction of any major weapons system. The environmental and reliability requirements for the final product are quite severe. In addition, the complexity of the finished item requires early detection of either actual or statistical trends which indicate faulty construction. Thus, this detection should occur at the subassembly level.

Many aerospace firms have turned to computer-controlled checkout systems as a solution to the problems of uniform testing, coherent data collection, and a usable retrieval system for their large scale checkout systems. For the Polaris program, Lockheed Missiles and Space Company (LMSC) devised ACRE[1], one of the original missile automatic checkout systems. More recently, several other automatic checkout systems have been developed such as those used for testing portions of the Saturn and Apollo vehicles.[2,3,4] The AutoSACE (Automatic Shorebased Acceptance Checkout Equipment) concept embodies several features which were shown to be desirable and necssary in these earlier checkout and evaluation systems.

The salient problem in the development of any large-scale system is the retention of sufficient performance flexibility to encompass unforeseen developments in the project. This problem is particularly acute in the development of a test system. Since the development of the package to be tested and development of the test system generally occur simultaneously, the design of the test system must maintain sufficient flexibility to encompass development difficulties in both projects. Therefore, the major processing element of the AutoSACE system is a network of stored program computers which analyzes data fed into the network by specialized test stations.

The range of AutoSACE testing includes the entire spectrum of checkout complexity ranging from elementary electronic modules to the integrated systems in the flight configuration. The test problem does not include, however, militarized checkout on the submarine or factory checkout of unassembled discrete components. The three areas of major significance where the tests will be performed are the LMSC factory at Sunnyvale, California; the Eastern Test Range at Cape Kennedy, Florida; and the depot at Charleston, South Carolina. Other testing is performed at vendor locations and remote development test areas. In all, about 40 different types of tests are performed by over 100 test stations.

The system

In establishing the configuration of the automatic test system, the two most controversial aspects were the processor hierarchy and the functional location of the breakpoint between the processor and the test station.

The question arises, when considering the factory test requirements, whether to have several autonomous test systems which are area-oriented or to have a totally integrated test system. A totally integrated system approach was chosen because:

(1) Simulation studies showed that an autonomous

area processor would bog down unless it had the support of an auxiliary memory. The cost of individual mass storage for each area processor would exceed the cost of a central processor with shared mass storage.

(2) A central processor can compile all test data from module test through system test, thus allowing rapid access to historical data and providing support data to system tests.

A second question arises when coupling the unit to be tested to the processor system. How can the interface with 40 unique packages be optimized? This has been a continuing problem with integral automated checkout systems. It has long been recognized[5] that the "universal" plus "adapter" philosophy holds great promise. Because time is usually critical in checkout system design, it is highly desirable to have universality to the lowest possible level, thus reducing unique design time. A building block construction of a complex checkout system lends flexibility and, yet at the same time, consistency to systems design. The use of universal sections allows a pyramiding of knowledge and provides a common interface for unrelated test functions.

The AutoSACE test station can be considered as two functional parts: a part unique to each package to be tested, and a part common to all test stations. This type of test station allows the processor to provide a universal interface to every test station.

The resultant system (Figure 1) is a two-level processor-oriented test and data collection system with a universal test station/processor interface. The system is integrated on a building block basis in the processors as well as in the test station digital control and measurement functions. In the total system there are 14 processors interfacing with over a hundred test stations. Consequently, great emphasis has been placed on processor/test station compatibility. The hardware that

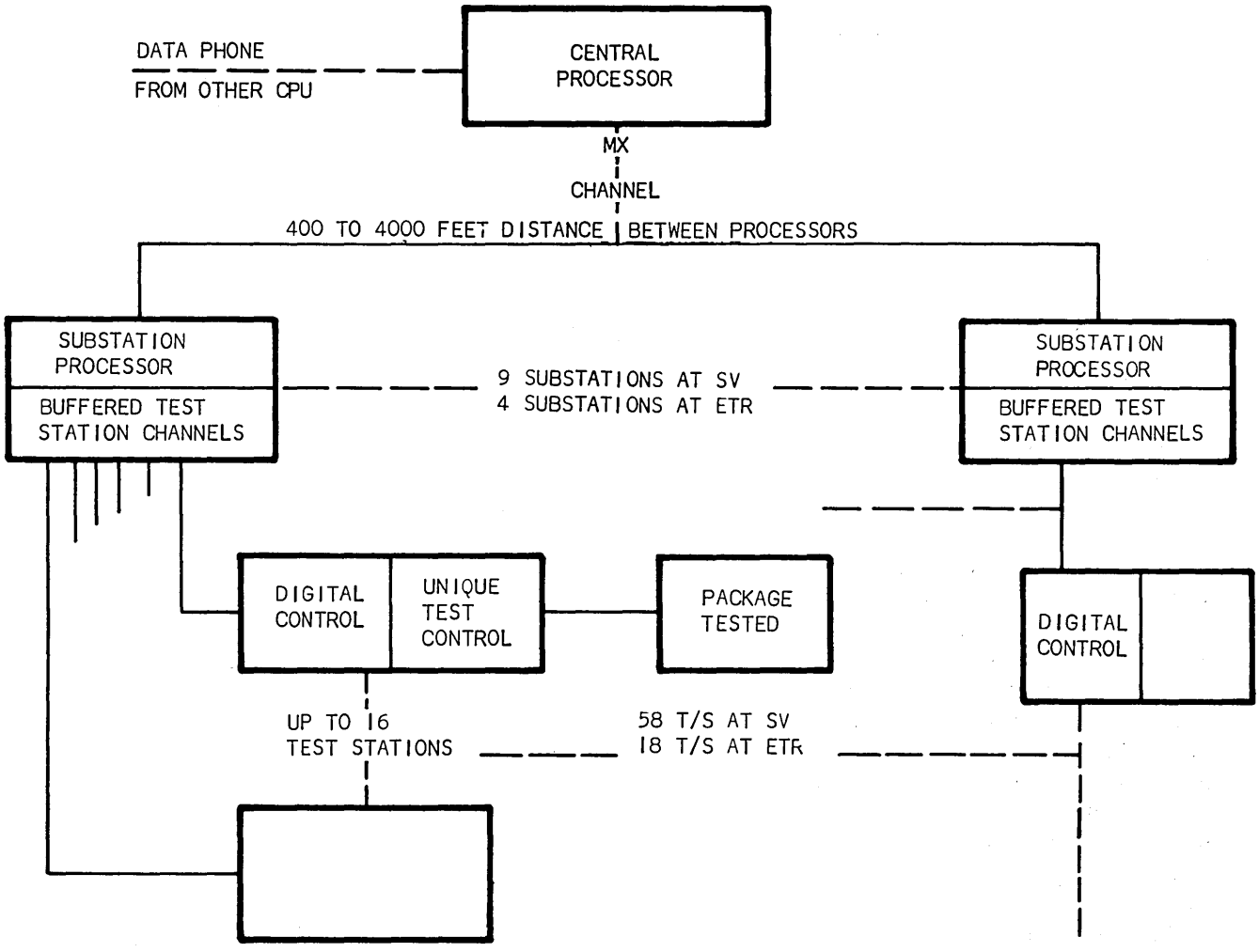


Figure 1—AutoSACE configuration

provides this compatibility is a modular Buffered Test Station Channel, common to all processors, and a Digital Control Bay, which is common to all test stations.

For AutoSACE, as with any system, certain requirements and restrictions force modifications and extensions beyond the most desirable operating procedures. Some of these are:

(1) The requirement to perform essentially the same tests at remote vendor locations as are performed in Receiving Inspection.
(2) The requirement for a backup mode for processor operation to allow continued testing if the processor becomes inoperative.
(3) The requirement for manual testing to perform special diagnostics and evaluation.

To satisfy these requirements, each test station is designed to operate autonomously in either a semi-automatic mode or a manual mode. In the semi-automatic mode, the processor control is replaced by a punched paper tape in the common Digital Control Bay. Output data is via a small printer also located in the Digital Control Bay. In the manual mode, control is accomplished with a 10-digit keyboard and data is displayed on indicators in the Digital Control Bay.

The processors

In defining the processor requirements, it was concluded that the test requirements at the depot do not warrant the two level processor system used at Cape Kennedy and at the factory. The central processor units (CPU) at both Cape Kennedy and the factory are equivalent in main frame capability but differ somewhat in peripheral capability. The central processor (Figure 2) is a 24-bit machine (as are all the processors) with a 32,000-word core memory having a two microsecond cycle time.

The CPU provides the central storage and distribution capability for the system. All test routines are stored in the mass-memory disc file and are provided on call, to the substation processor and then to the test station. All test data received from the substation is formatted, collated, and stored. Selected test data can be stored temporarily in mass memory, displayed on a CRT at the test station, or printed by a line printer. Permanent storage of test data is done on magnetic tape.

Non real-time functions performed by the central processor include computation, program development, and listing of test programs and results. Computations

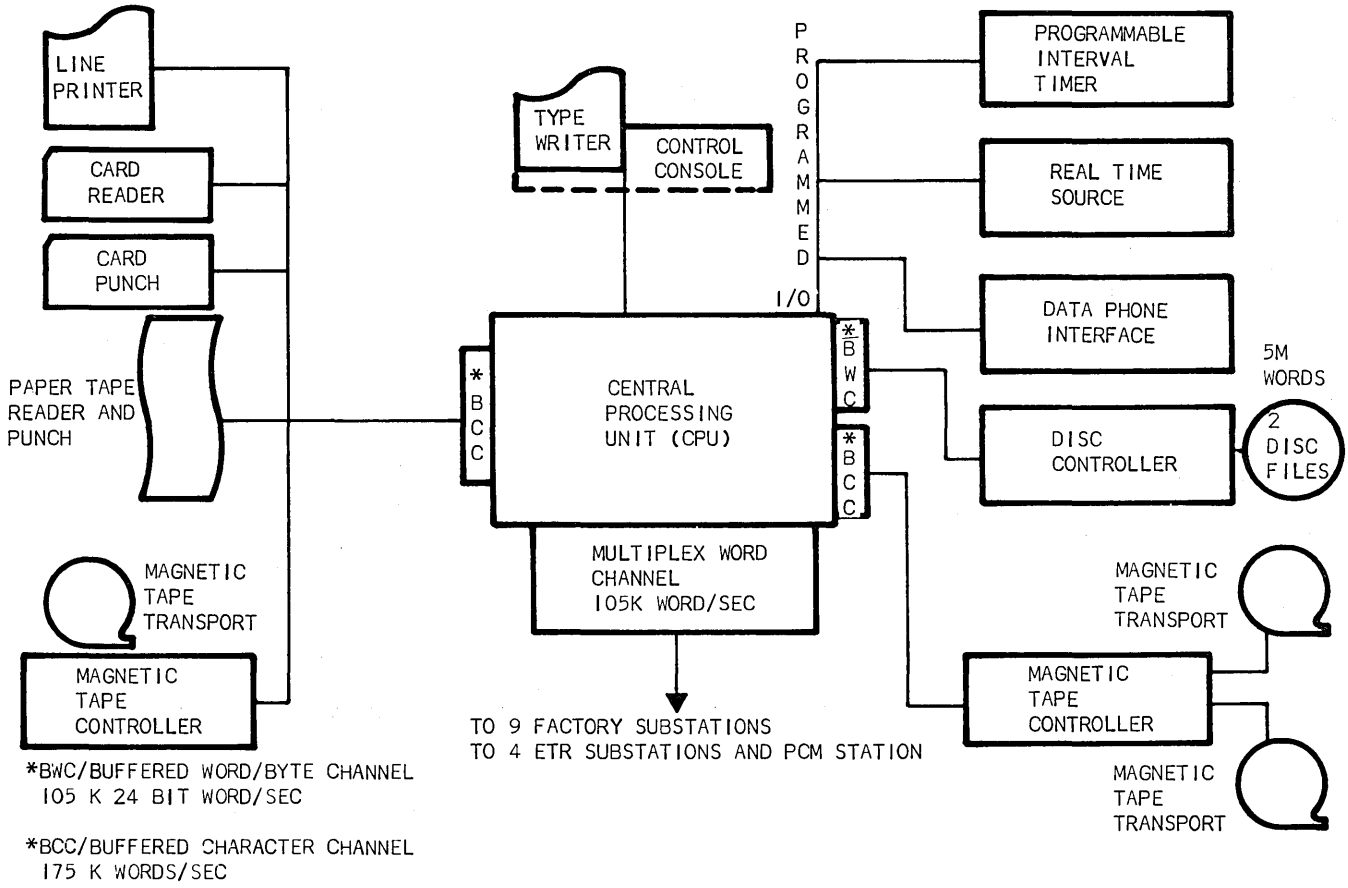


Figure 2—Central processor configuration

include arithmetic, logical and statistical functions such as value averaging, fault isolation, and failure analysis and prediction.

The CPU contains the Fully Automatic SACE Translator (FAST), a unique problem-oriented language compiler for translating source language programs and test routines into machine language. Access to the compiler is from either the CPU or the test station CRT. At a distance of 400 feet to 4000 feet from the central processor, the substation provides the control and sequencing link between the CPU and the test station. Data is transmitted between any substation and the CPU in a half-duplex, bidirectional mode in six-bit characters (Bytes) at rates up to 105,000 characters per second. All data is transmitted via differential current mode circuits and parity is checked on all reviewed data. Communication between CPU and substations is controlled by service interrupts from both ends of the link, but in conflicts of transmission the substation always has precedence.

The mainframe of the substation houses a 16,000-word core, which, like the CPU, is 24-bits per word. A typical substation processor system is shown in Figure 3. Actually, processor peripherals vary according to the unique requirements of each area.

The substation has the necessary interrupts and controls to provide test station mode control for either test control or data display on the CRT. As the result of a test station request, the substation will, in turn, request appropriate programs and routines or data from the CPU for either display on the CRT or print out on the typewriter. Test instructions and addresses are transmitted to the Digital Control Bay in a parallel coded format. Test data received at the substation is separated, formatted, and transferred to the central processor for storage and output. Test data will be analyzed on a limited basis as a part of the test program; a typical analysis being limit comparison on a GO/NO-GO basis.

The Buffered Test Station (BTS) Word Channel, a part of each substation processor, is a special unit which allows communications with up to 16 remote test stations on a parallel word multiplex basis. Each sub-channel is divided into two sub-channels for input and output to allow full-duplex bidirectional communications. At present, software allows simultaneous com-

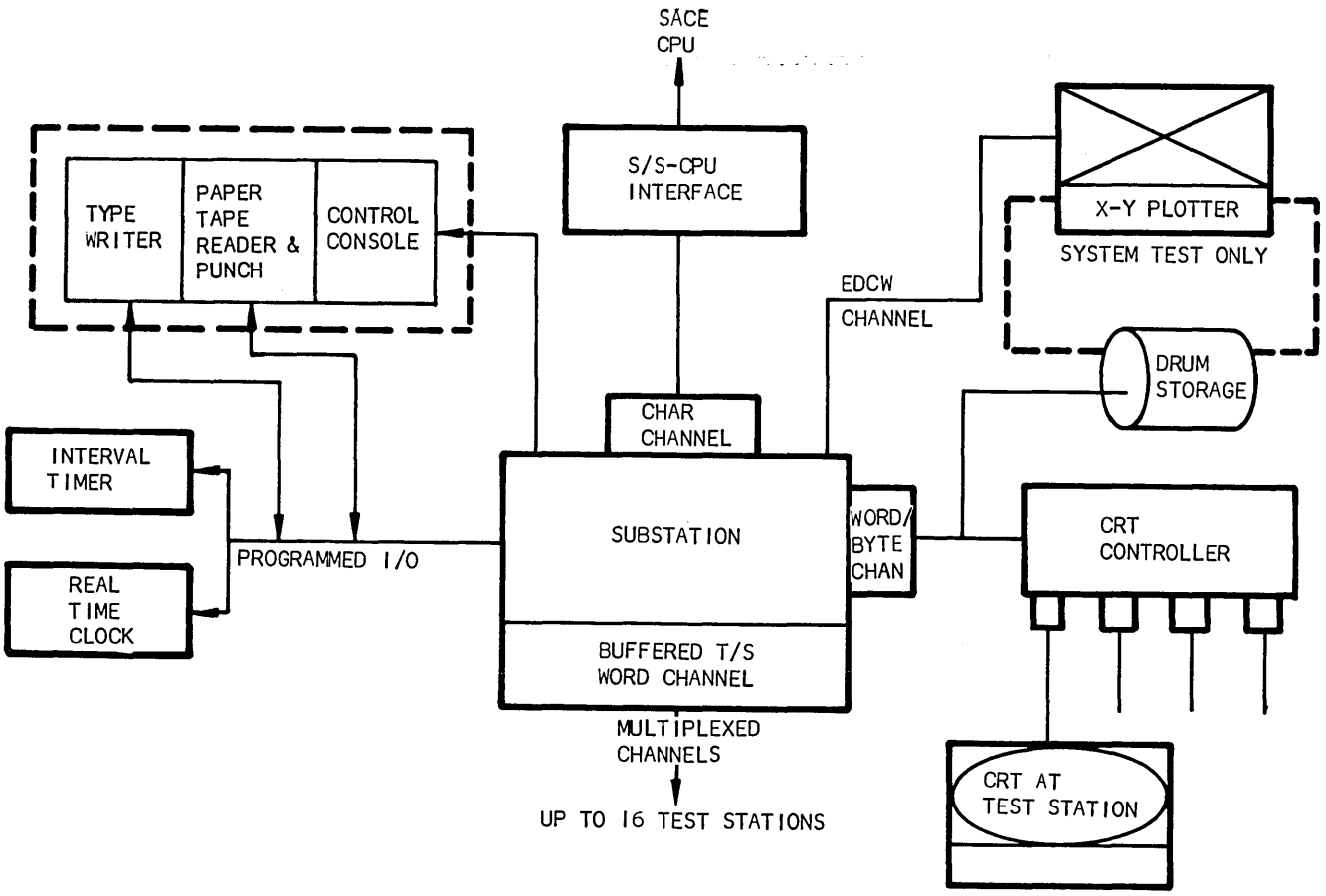


Figure 3—Substation configuration

munications with two test stations at a cumulative data rate of 105,000 words per second.

Like the CPU interface, the BTS Word Channel uses low level DC current mode differential drivers and receivers to communicate with test stations up to 1000 feet away. The interface is so designed that test stations, when not on line, can be disconnected from one sub-channel and moved to another sub-channel without affecting operations of other on-line stations.

A similar interface exists between the CRT Controllers and the CRT located at the test station. Because there is a ratio of about three test stations for each CRT, it is necessary to move the CRT from station to station without affecting other stations. This capability is provided by division of the controller memory and by switching the output or providing multiple lines to a test station. The technique used depends upon the area. Any switching is controlled by the processor as a result of test station data inputs.

The SACE test station

The elemental requirements for any test station are shown in Figure 4. The Digital Control Bay (DCB) must, in effect, present itself to the substation as a computer peripheral device for both input and output of information. A typical test station is shown in Figure 5 with a package under test and the operator seated at the CRT. The function of the character-generating CRT is to provide the man-machine interaction for

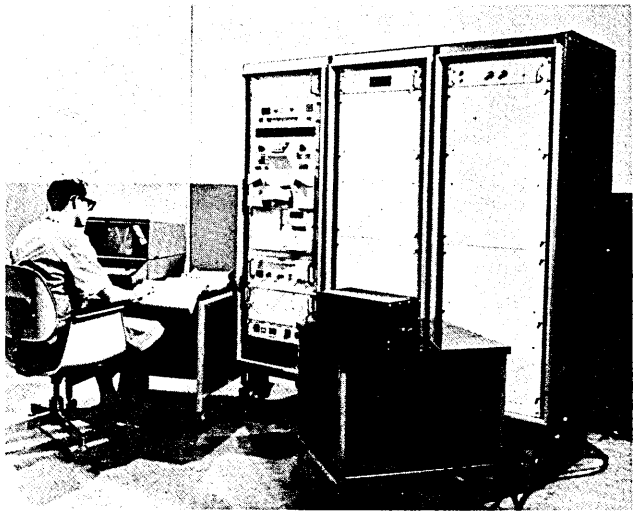


Figure 5—AutoSACE test station

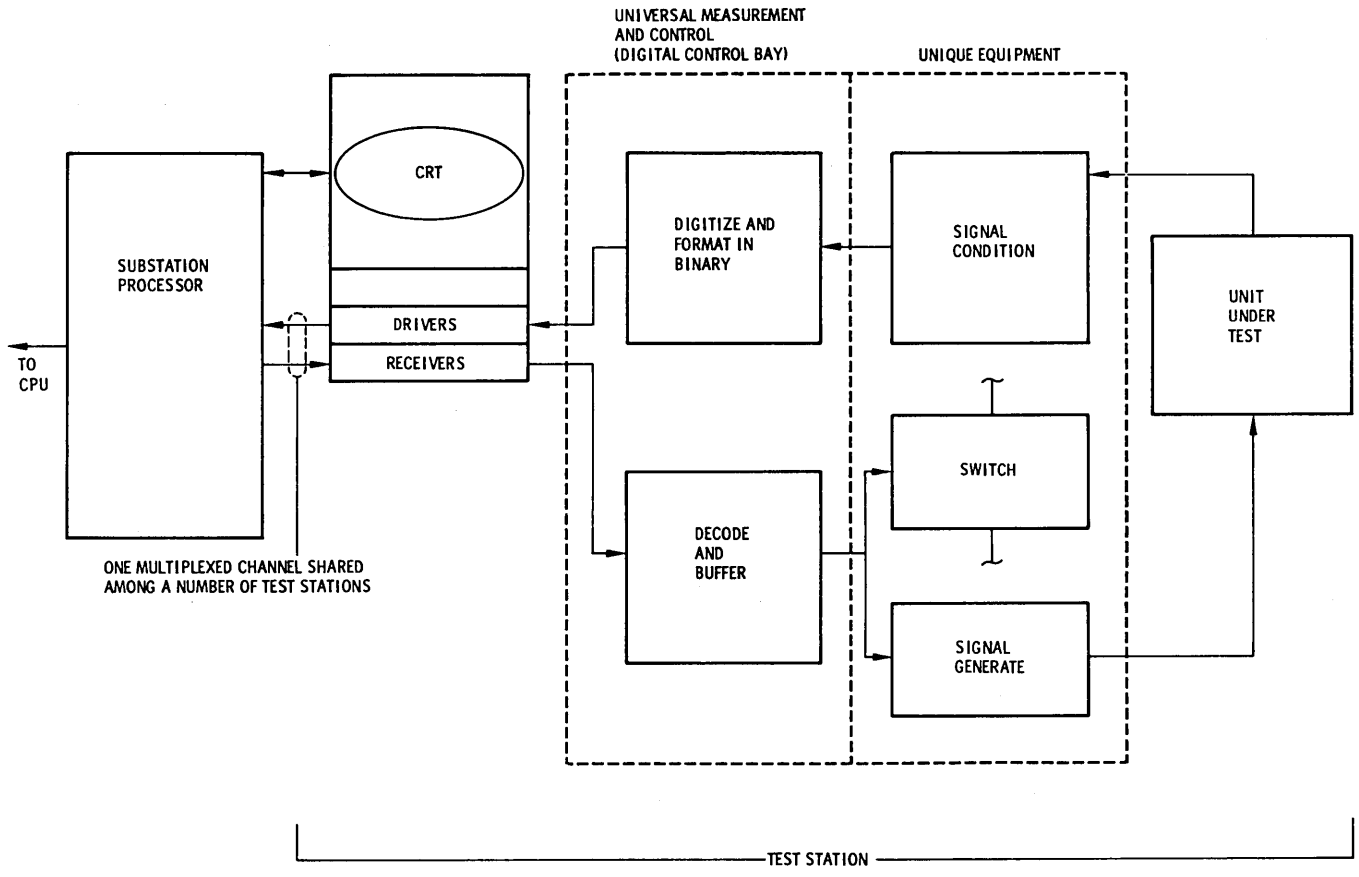


Figure 4—Test station elements

test control and data display. For any checkout system, the CRT display is a highly flexible ingredient which finds a multiplicity of uses, such as:

(1) Test Initiation
(2) Test Directions for the Operator
(3) Test Program and Data Display
(4) Program Alteration

A significant advantage to the use of CRTs is that a large amount of matter may be displayed for the operator without imposing an undesirable load on the substation. This permits maximum use of the substation for the actual test problem and allows the operator to selectively instruct only the printout of that material in which he has an immediate interest. A small control panel on the CRT table provides the operator with access to the processor interrupt system. The panel enables him to start and stop the test at the push of a single switch.

The basic control address instruction, as used in all three modes of test console operation, is a 15-bit word which contains two fields, a 7-bit "location" code and an 8-bit "control" code. The location code is used to specify one of 128 control buffer registers. The 8-bit control code portion of the control address is then set into the specific register. The octal control address format for the processor mode is shown below:

Decode (echo check) bit	7 bit location							8 bit location							
	L7	L6	L5	L4	L3	L2	L1	C8	C7	C6	C5	C4	C3	C2	C1
24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Bit 24 is used with a substation routine to indicate whether the last control address sent should be decoded and replaced with the next or only replaced with the next control address and not decoded. In this manner, a control address may be transmitted to the DCB, registered there, and sent back through the data lines to the substation for comparison and verification. Upon verification, the substation sends the next control address with the decode bit set. This causes the last address to be decoded and the new one to be registered, etc. This method requires additional time because it must be done as single-word transmissions, as opposed to buffered transmissions. Because this mode of check is under program control, only critical control addresses need be checked.

A comprehensive monitor is maintained on all transfer control lines between the substation and test console. This is an essential feature of any automatic checkout system since in most instances the operator cannot directly recognize when hardware is hanging up the program or vice versa. In this multi-user system, transfer control line monitor logic is utilized to sense test stoppage and signal the processor through the interrupt system.

The digital control bay

In order for the Digital Control Bay to find multiple use in test consoles ranging widely in complexity, it must by necessity be universal and flexible. Furthermore, it must provide all test control in two non-computerized modes of usage.

Figure 6 represents a functional block diagram of the DCB. Control address instructions are received in the entry register (E register) and decoded from there. The decoding process causes the location code to be set into one of 128 separately addressable integrated circuit flip flop registers (locations) of the control buffer register (B register). Of these 128 registers, approximately one third are used for DCB self-control and the other two thirds provide control for unique stimulus generation and switching.

The Digital Control Bay has the capability of accepting signals for measurement in either analog voltage or digital form. The analog inputs are channeled to an accurate, high speed analog-to-digital converter. Any other signal conversion will be performed in an adjacent unique bay. Parameters to be converted within the unique bay may include frequency, resistance, pulse width, and AC. The conversion may use functions provided by the DCB, but the resulting signal to be measured must be in analog, BCD, or binary form.

All DC voltage measurements will be made by a multi-channel multiplexer and an automatic ranging analog-to-digital converter with a 13-bit data output. The design is a modification of standard commercial equipment to meet AutoSACE universal measurement requirements, and to minimize signal conditioning.

Digital inputs are handled by a digital selection matrix, with each device input for 4-BDC or 4-octal (in the case of binary inputs) characters plus two range bits, a sign bit, and strobe lines. A good example of the usage of these inputs is the measurement of frequency by any commercially available counter which can provide digital output lines.

SACE system requirements place heavy emphasis on continuous monitoring of discrete-event and time-oriented functions. An event is defined as a transition in either the positive or negative going direction of a single digital line.

An important feature of the DCB is its event register, which provides a fundamental capability to monitor simultaneously up to 15 discrete-event line inputs.

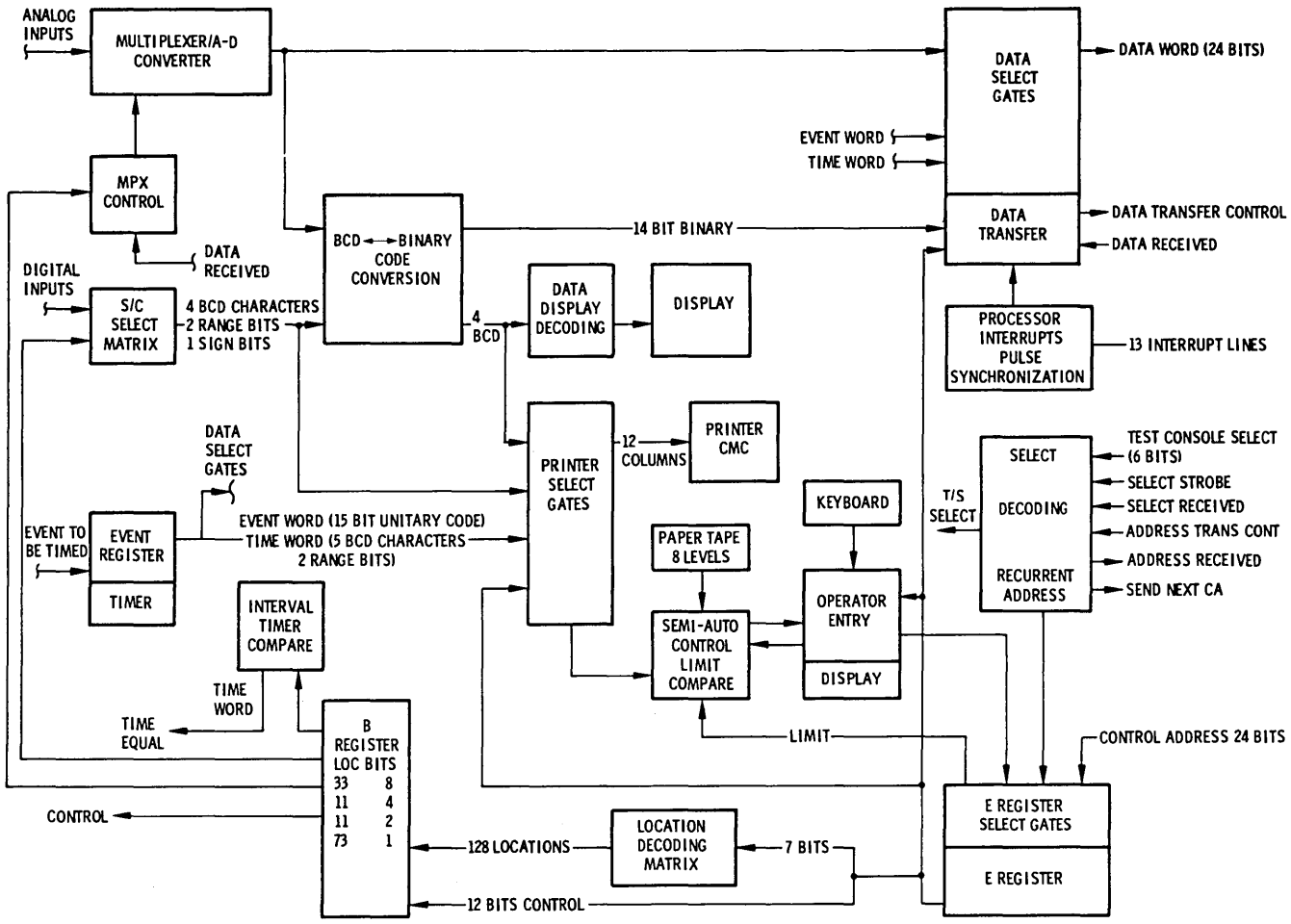


Figure 6—Digital control bay functional block diagram

A hybrid sensor or signal conditioner will convert a parameter from one signal form to a step function which is electrically compatible with the monolithic integrated circuit requirements of the event register. For example, the event might be the result of an integrated circuit analog comparator which is used to sense DC amplitude and trigger at a set level. Other sensors trigger on frequency, pulse width, the presence of electrical noise, the presence of an AC wave form, and amplitude of an AC wave form. Some examples of these transformations are shown in Figure 7.

The sensor circuits, in conjunction with the logic level change detection network, provide the processor with a form of data compression of event information. When any one of the 15 sensors is triggered, the level change detection circuit causes the status of all 15 event lines and the time of occurrence to be registered.

A priority interrupt line is utilized to signal the substation that an event has occurred. In this manner monitoring analog signals on a continuous basis for abnormal conditions, can be accomplished in numerous ways simultaneously with normal test operations. Microsecond capabilities of the register/timer scheme, along with a high speed A to D converter, provide simple and flexible tools for the solution of continuous monitor, fault isolation, and quality control problems.

Data formatting

Conversion logic must provide a decimal display for the manual and semi-automatic modes and send the same information in binary in the processor mode to the processor.

Choosing a multiplexer and analog-to-digital converter with a 5-volt and 40-volt full scale range, allows one section of logic to be used to convert both directions. For one device to perform both functions when using the shift and add/substract method, some limitations must be placed on the numbers to be converted, namely:

(1) BCD numbers to be converted to binary must

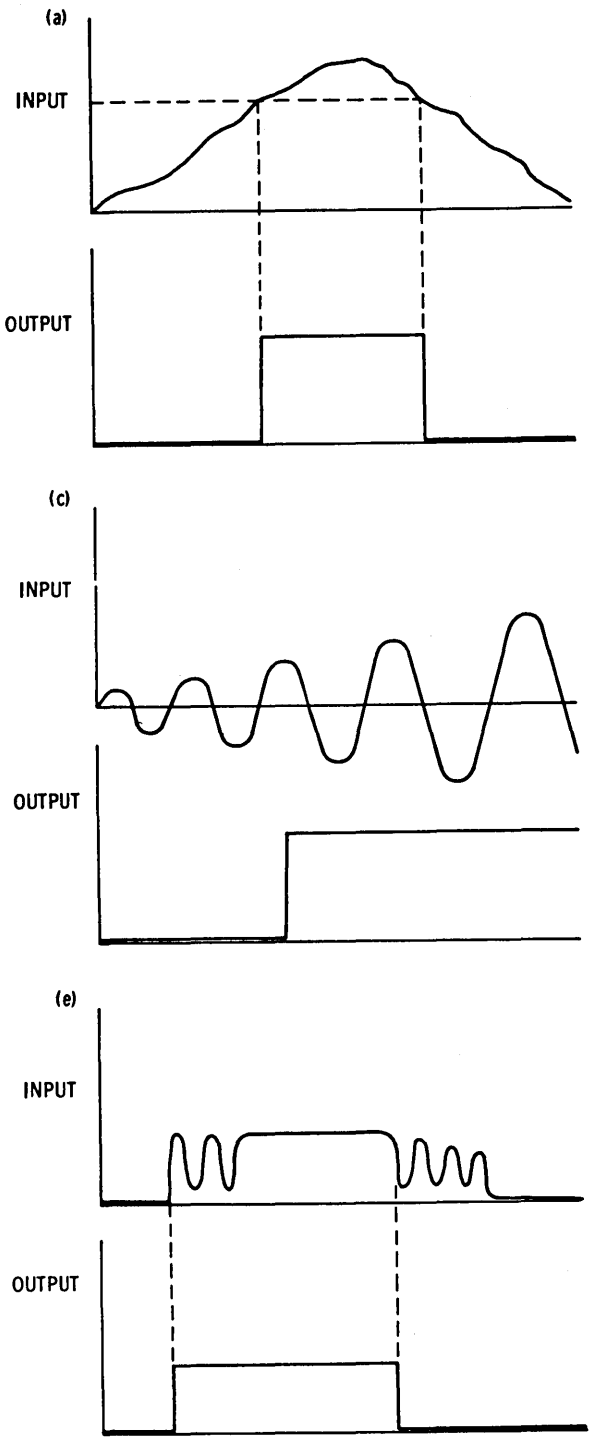


Figure 7—Response curves for typical sensors

be considered as whole numbrs; e.g. 1.279 seconds must be thought of as 1279 milliseconds because the BCD least significant bit must equal the binary least significant bit.

(2) Binary numbers to be converted to BCD must be considered as a fraction of full scale; e.g., 1000 . . . is one-half of full scale and the binary full scale must equal the BCD full scale. In general, full scale may be any value equal to $2^N \times 10^M$, where N is any integer, positive or negative.

The shift and add/subtract alogrithm was chosen because it entails a minimal amount of logic for conversion in both directions and does not severely restrict the design of the analog measurement subsystem.

Software

As mentioned earlier, the controversy involving processor hierarchy and interface definition resulted in a compromise of views to allow system optimization. Similarly, the controversy over the degree of software complexity resulted in compromise. The choice, simply stated, was whether or not to buy more hardware in order to reduce program complexity. Although the number of processors could be minimized by more intricate time-sharing or multiprogramming, under the best conditions, software development can be a predominant part of the system cost[6]. In addition, some contemporary systems had been bogging down in multiprogramming. The resulting decision was to provide sufficient hardware to allow considerable flexibility in development of software to give added insurance that the system will be operating on time. The hardware also gives additional expandability that a program-saturated system could not give.

The system software developmnt is thus directed toward performing test program oriented functions and providing housekeeping and interface control. The problem-oriented software is built around the test requirements on one hand and the standard machine language on the other. A program is initiated from test requirements which are written as an Acceptance Test Procedure (ATP). This ATP is transferred into an abbreviated English language version which in turn is transformed into machine language via a P.O.L. translator. The P.O.L. language, developed for C3 testing, is called Fully Automatic SACE Translator (FAST).

FAST

For the designers who prepare the test procedures, the Fully Automatic SACE Translator provides a processor entry tool that (1) reflects the flexibility of the processor software and (2) provides a test procedure oriented language that the designer can more readily understand. FAST was developed from a background of ATOLL,[1] the Saturn translator. It is

not intended to be a natural language panacea for programming problems, but it is intended to give the designer a tool that allows the manipulation of familiar test station oriented parameters rather than computer language. The worksheet for FAST programming (Figure 8) is a standard 80-column worksheet broken into four major sections. Columns 6 and 7 provide the sequence number of the item on the sheet relating to a single step. Columns 9 through 15 provide the operations or instructions. About 40 instructions can be grouped into four categories:

I. Operators that prepare the test station for measurement, such as SEND commands to the test station, SEND-CM.

II. Operators that implement the program, such as CALL Subroutine operation and prepare return linkage, CALL-SB.

III. Operators used for measurement of a test point, such as MEASure DC voltage and react, MEAS-DC.

IV. Operators that modify the location or value of data, such as MOVE one Data Word from a source location to a final location, MOVE-DW.

Columns 17 through 72 describe the details relating to the operator. Operands are grouped to the left to indicate such information as control address to the test station, i.e., 024003, 025014 and 000012. Notes relating to the operator and operand are located to the right, i.e., measure PS1 on Channel 012 of the multiplexer and compare with limits specified in the table. Columns 73 through 80 are used for card sequence numbers for sorting. Because of the potential cost of programming and operating time due to redundant operations, many functions that are repetitive are put in tables for reference. This information may be in terms of limits, switching addresses to be sent to the digital control bay, messages to be displayed in the CRT, or time values. In all, there are eleven types of tables. An example of the use of the table is given in Figure 8, where the value of the power supply reading is compared to the limits given in TABL substep 04.

A brief example using tables for a set-up and comparison is also shown in Figure 8. Substep 00 enters the subroutine RMPG and performs a remote test program that sets up for resistance and continuity measurement. Substep 02 sends the control addresses 024003 and 025015 to the test station to light the specified indicators and to set a power supply to 33

TP SEGMENT | TP REVISION | BLDG | FAC | TEST PROCEDURE NUMBER | DATE OF REQUEST | TEST CONSOLE NUMBER | TP NAME | CCPS CONTROL NO. | DATE NEEDED | PRIORITY
TEST WRITER | CODE | PHONE

PAGE ____ OF ____

FAST PROGRAMMING WORK SHEET

STEP	SS	OPERATOR MAJOR	OPERATOR MINOR	GENERAL OPERANDS, COMMENTS	P3	PGM		CARD SEQUENCE
TABL	00	LIMT	DL	+17.45 ,+17.55 ,1.6 ,VDC	1	000	ØUTPUT	FUNC2114
TABL	01	LIMT	DL	+19.42 ,+19.53 ,1.6 ,VDC	2	002	VØLTAGES ØF	FUNC2115
TABL	02	LIMT	DL	+19.45 ,+19.55 ,1.6 ,VDC	1	002	PS1 AND PS2	FUNC2116
TABL	03	LIMT	DL	+27.64 ,+28.36 ,1.6 ,VDC	2	012	WITH VARIØUS	FUNC2117
TABL	04	LIMT	DL	+32.95 ,+33.06 ,1.6 ,VDC	1	012	PRØGRAMMING	FUNC2118
TABL	05	LIMT	DL	+34.03 ,+35.08 ,1.6 ,VDC	2	014	INPUTS	FUNC2119
				---RESISTANCE AND CØNTINUITY TEST				
0001	00	CALL	SB	RMPG ENTER AND EXECUTE REMØTE PRØGRAM TEST				
0001	01	SEND	CM	024003,025014, LITE CØNT. AND RESISTANCE LEGEND				
0001	02	DLAY	XE	3.0 AND PRØGRAM PS1 FØR 33VDC				
0001	03	NDEX	ST	0,4,I1 SET I1 TØ 4				
0001	04	MEAS	DC	000012,LIMIT(I1),D,ERP1 MEASURE PS1, CØMPARE WITH LIMITS				
				SET IN THE FIFTH ENTRY ØF TABLE				
				"LIMIT", DØ NØT SAVE THE RESULTS				
				BUT IF IN ERRØR GØ TØ "ERP1"				

Figure 8—Example of FAST worksheet

volts. Substep 03 sets up index register number 1 to the value in table 04. Substep 04 measures the 33-volt supply on Channel 12 of the multiplexer and compares it with the limits set up by substep 03. If the measured value is out of limits, an error routine ERPI, will be entered.

Software allocations

When considered on integrated basis, the processor system has an extremely powerful software handling capability. Four major areas for handling programs and data are shown in Figure 9. All operation involves an interplay of software between these four areas.

The mag tape units and the disc file serve as permanent mass memory, with most programs emanating from one of them and all data ending up in one of them. To reduce the complexity of control, each of the three mag tape units are dedicated primarily to a single function. However, the disc file at one time or another handles most programs and data. About one-half of the 5 million words is dedicated to temporary storage of test data. About 1 million words contain the acceptance test procedures for all stations. The remaining disc space provide permanent storage for the processor executives, special test data, CRT forms, and fault isolation directories.

The CPU mainframe core is split between the processor oriented executive programs and the buffer areas that handle ATPs, test data, and formats. In both levels of processors, considerable advantage was taken of existing software, but one portion that required extensive development was the interface drivers. This is true of both the CPU/substation interface as well as the substation/test station interface. As with the CPU, a major portion of the substation memory is dedicated to ATPs. About 2000 words of the total 16,000 is dedicated to storage of FAST operator subroutines. Equal amounts are also shared by the substation executive, and test data collected from the

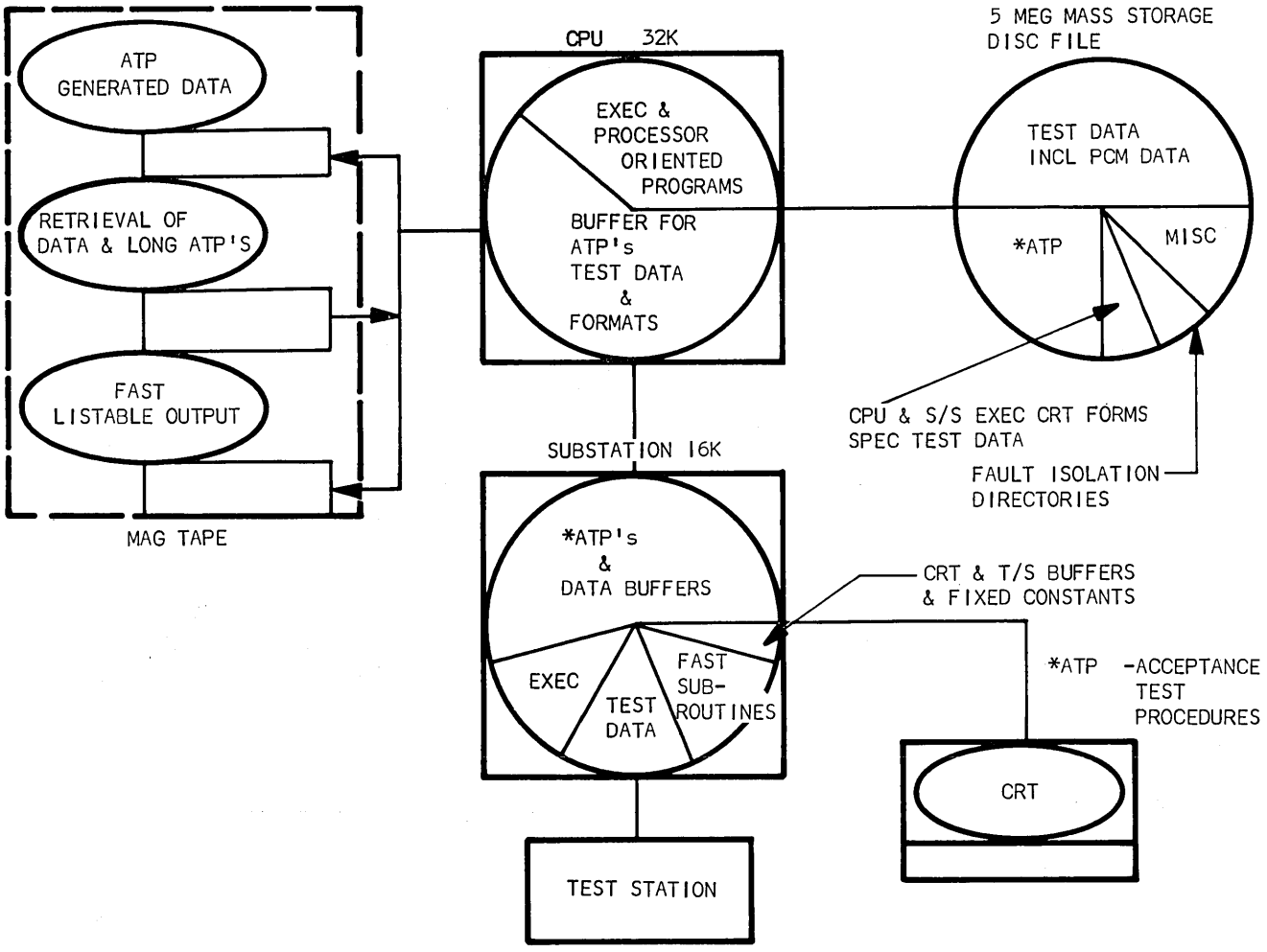


Figure 9—Software allocations

test station. Locations for test data related to specific ATPs are reserved by the ATP. Buffer locations for the test station and CRT interface take about 7 percent of the total space. In general, the substation mainframe memory is hard pressed for locations, and as a result, some substation-oriented software is located in the disc file. This includes CRT forms, special test information, and fault isolation directories.

CONCLUSION

AutoSACE provides a high capacity test and data collection system with the ability for rapid analysis and dispensing of results for the enhancement of Poseidon effectiveness. Features provided by this system include:

(1) Standard processor system
(2) Universal Interface
(3) Accurate, high-speed measurement and control
(4) Building Block Design
(5) CRT input/output for the operator
(6) A unique problem oriented language

Although this paper has emphasized the testing of complex electronic assemblies at fairly low rates, the same features could be applied to problems demanding higher rates — for example, factory testing or integrated circuits. Considerable time and expense might be saved if test data on first run samples could be quickly analyzed to determine trends that indicate weak design or weak material characteristics.

ACKNOWLEDGMENT

Space limitations make it impossible to acknowledge the efforts of all those who have contributed to this program. Special thanks, however, are due Hugh Underwood, of LMSC's SACE Engineering Department, whose knowledge of the system and critical comments have helped greatly in the preparation of this paper.

REFERENCES

1 R A KIRKMAN
The use of automation in the checkout of aircraft, spacecraft, and missile systems
Presented to AIAA Annual Meeting Washington D C 29 July 1964

2 R V MURAD J F UNDERWOOD
Automatic checkout for Apollo
Presented to IEEE International Convention on Military Electronics, 14 September 1964

3 L T MAST *et al*
Survey of Saturn/Apollo checkout automation,
RAND RM-4785-NASA; NASA CR-74349, X6616385 January 1966

4 H HESKIN
Saturn checkout system
vol 10 no 12 December 1961

5 J Q MALOY
Advanced automatic checkout equipment
presented to ART Ground Support Equipment Conference, Detroit, Michigan, 23 March 1960

6 V MAYPER
Programming for automated checkout
1, April 1965; Part 2, May 1965

7 B L RYLE
The ATOLL checkout language

Real-time spectral analysis on a small general-purpose computer

by ARVID G. LARSON and RICHARD C. SINGLETON
Stanford Research Institute
Menlo Park, California

INTRODUCTION

There is a growing need for methods of quickly estimating the changing frequency content of a nonstationary signal. In this paper we describe a method for doing spectral analysis in real time on a small general-purpose digital computer, and discuss some of the theoretical and practical problems of developing similar systems for other computers.

Until recently, the field of real-time spectral analysis has been left to analog equipment. The relative slowness of digital methods did not allow the spectral bandwidth required for most practical problems. However, digital computers have grown progressively faster and cheaper. In addition, the recent development of programs for the fast Fourier transform allows a major reduction in the time for transforming a data sample from the time domain to the frequency domain. Digital analysis can now be done in real time with useful bandwidths, and in many cases offers important advantages over analog methods. High dynamic range is more easily attained; once the data has been sampled, the subsequent digital computations can be carried out with high precision. With digital methods, the number of possible resolution cells in the frequency domain is limited only by the data-storage capacity of the computer; the problems of building narrow-bandwidth filters are avoided. Also avoided are the problems of gradual drift of response characteristics. Furthermore, as compared with a special-purpose digital or analog analyzer, a programmed general-purpose computer allows greater flexibility in the analysis method.

The methods of real-time digital spectrum analysis are particularly applicable in the area of process control. Among other advantages, it is possible to compute spectrum analyses of several analog signals at the same time on one analysis system, producing simultaneous spectral estimates for each input channel. A number of inputs can be kept under surveillance and the allocation of the available bandwidth capacity changed in response to abnormal conditions. The computer can in addition perform the subsequent computations associated with the process control.

There are a variety of potential applications for a real-time analysis system. Examples are air traffic control using Doppler radar techniques, Doppler radar measurements of meteors or terrestrial orbital objects, vibration measurements during non-destructive testing, and medical monitoring during intensive care or surgery. In many other applications rapid availability of results from a number of data sensors is needed for efficient operation, as in seismic exploration.

Real-time digital spectral analysis can be done on any computer having the necessary provisions for accepting a stream of input samples as they become available. But in order for the system to be of practical interest, the computation must be fast enough to give sufficient spectral bandwidth to meet the requirements of the intended real-time application. We have programmed a real-time digital spectral analysis system on a Scientific Data Systems (SDS) 930 computer, using machine language to gain the speed advantage of fixed-point arithmetic on this machine and to allow use of input interrupt control. Computing speed permits a maximum continuous sampling rate of approximately 2000 Hz, and thus a maximum bandwidth of 1000 Hz. This system is being used to process Doppler radar data. The results obtained have demonstrated the feasibility of real-time spectral analysis on a digital computer.

Analog-to-digital conversion

The first step in the digital analysis of an analog signal is to sample the signal by means of an analog-to-digital converter (ADC). In doing this we select a

sampling frequency f_s of at least twice the maximum frequency of interest in the subsequent analysis. To allow use of the fast Fourier transform algorithm, sample values are equally spaced in time.

In the subsequent analysis of the digital values, we estimate the power spectrum from 0 to $f_s/2$, a bandwidth of $f_s/2$. Any signal or noise power at frequencies beyond $f_s/2$ will also be folded onto this interval with

$$0 \equiv f_s \equiv 2f_s \equiv 3f_s \equiv \ldots$$

Blackman and Tukey[1] discuss this property, calling it *aliasing*. Unless the original input is known to contain negligible power at frequencies beyond $f_s/2$, a filter network should be included ahead of the analog-to-digital converter to attenuate power above $f_s/2$. If the filter also attenuates the signal below $f_s/2$, the computed spectrum can be corrected by dividing by the power transfer function of the filter.

In using an ADC, we must scale the input to remain within the fixed amplitude range of the converter. Otherwise the signal will be clipped, with consequent distortion. In our present system we set the input level at the beginning of an analysis and leave it fixed. A possible improvement would be to provide means of changing the input level between successive groups of sample points, perhaps with the computer controlling the setting; this would allow more nearly maximum use of the available converter resolution.

Frequency shift

Sometimes we wish to analyze a limited band of frequencies displaced from zero, rather than the fundamental interval $[0, f_s/2]$. The aliasing property leads to a useful technique of doing this. With a sampling rate of f_s, any interval $[kf_s/2, (k+1)f_s/2]$ between two integer multiples of the bandwidth $f_s/2$ can be selected for analysis. The signal power is limited to the desired interval by use of a bandpass filter, and then the sampling and analysis are done as for the interval $[0, f_s/2]$. The frequency scale of the estimated power spectrum is determined as follows: if the analysis interval starts on an even multiple of $f_s/2$, then $kf_s/2$ corresponds to 0 and $(k+1)f_s/2$ corresponds to $f_s/2$; otherwise $kf_s/2$ corresponds to $f_s/2$ and $(k+1)f_s/2$ corresponds to 0, with the spectrum arranged in decreasing frequency order. Although choice of analysis intervals is limited to consecutive integer multiples of $f_s/2$, the sampling rate can usually be adjusted to give a close approximation of a desired interval $[f_1,f_2]$ if f_2-f_1 is less than the maximum bandwidth of the system. As compared with frequency shift by the heterodyne method, this technique avoids possible distortions due to the added steps of mixing the input and a reference signal, and then low-pass filtering prior to sampling. However, the aperture time of the ADC limits the upper frequency at which this approach can be used. One application for this technique is in the observation of Doppler shifts about a radar carrier frequency. Since in this case we wish to distinguish positive and negative deviations, the carrier should be near the center of the analysis interval and not at one of the end-points.

Digital spectral analysis

The signal to be analyzed is now in the form of a sequence $\{x_j\}$ for $j = 0,1,\ldots n-1$ of equally spaced data points, with n an even number (some of the expressions change slightly for odd n). Our objective is to estimate the power spectrum of the process giving rise to this sample. We start with data in the time domain and end with a smoothed power spectrum in the frequency domain. The intervening computational steps can be done either in the time or frequency domain, as illustrated in Figure 1, and the point of crossing over to the frequency domain can be chosen on the basis of computational convenience. Numerically equivalent results can be obtained following any of the possible paths in this figure.

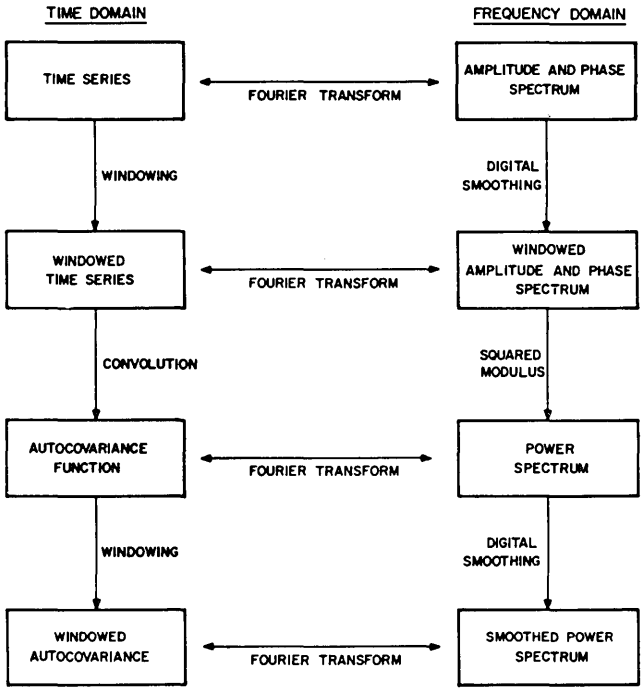


Figure 1 – Time-frequency relations in digital spectral analysis

In our analysis we use methods similar to those suggested by Bingham, Godfrey, and Tukey.[2] Taking advantage of the speed of the fast Fourier transform algorithm, discussed in the next section, we transform at the first or second level of Figure 1, avoiding the relatively time-consuming convolution operation to compute the autocovariance function in the time domain.

The major time element in a spectral analysis calculation is time for computing the Fourier transform. The reader familiar with the method proposed by Blackman and Tukey[1] and other writers prior to the fast transform will recall that the usual practice has been to compute an autocovariance function estimate

$$\psi_\tau = \frac{1}{n} \sum_{j=0}^{n-1-\tau} x_j x_{j+\tau} \qquad (1)$$

for $\tau = 0,1,\ldots T$, with the maximum lag T typically in the range of 10 to 30 percent of n, and then to compute the estimated power spectrum with a Fourier cosine transform

$$P_k = \frac{1}{T}\left\{w_0\psi_0 + 2\sum_{\tau=1}^{T-1} w_\tau\psi_\tau \cos(\pi k\tau/T) + w_T\psi_T \cos(\pi k)\right\}$$

for $k = 0,1,\ldots T$, where w_τ is a window function for the purpose of reducing interactions of spectral estimates. As compared with transforming the original data, the size of the transform is reduced by a factor of T/n by truncating the autocovariance function, a form of windowing, and by an additional factor of two due to the autocovariance being an even function. Referring to Figure 1, we see that in this method of analysis the transformation from the time domain to frequency domain is done at the final level.

Use of the fast Fourier transform algorithm reduces the number of arithmetic operations for the Fourier transform from the order of n^2 to the order of $n \log_2 n$. This improvement has led to revisions in computing methods in spectral analysis. The convolution operation in computing the autocovariance function, requiring

$$n + (n-1) + \ldots + (n-T) = n(T+1-T(T+1)/2 \approx nT$$

multiply and add operations for n data points and a maximum lag of T, is now in most cases the slowest link in Figure 1. We avoid this link by transforming to the frequency domain at the first or second level, in advance of the convolution. If we want the autocovariance function, we get it by computing the inverse Fourier transform of the power spectrum. As discussed by Stockham[3] and others,[4,5] the fastest route to the autocovariance function is now usually through the Fourier transform and inverse, yielding the power spectrum as a by-product.

In computing the Fourier transform of a sequence of length n, the sequence is treated as a portion of an unending sequence of period n. Thus in computing an autocovariance function estimate by the Fourier transform method, the n sample values can be viewed as arranged in a circle, with x_0 following x_{n-1}. Instead of dropping the end values after each shift in computing lagged products, as in Eq. (1) above, the end values multiply the initial values as follows:

$$\psi'_\tau = \frac{1}{n} \sum_{j=0}^{n-1} x_j x_{(j+\tau) \bmod n} \qquad (2)$$

for $\tau = 0,1,\ldots n/2$. If the original signal is non-stationary or even periodic of some period other than n, the mismatch of the two ends leads to an undesirable level of interaction of spectral estimates and a serious loss in dynamic range of the analysis. One way of improving the situation is to lengthen the data sequence with T zeros; the autocovariance computed by the transform method then agrees out to a maximum of T lags with that computed by Eq. (1). A better way is to truncate the time series gradually, as proposed by Bingham, Godfrey, and Tukey.[2]

In our real-time analysis system we use the data-window function

$$w_j = \sin^2(\pi j/n) \qquad (3)$$

for $j = 0,1,\ldots n-1$. (When greater reduction in interaction of spectral estimates is needed, we use the second or third power of this function.) The windowed time series

$$x'_j = \sin^2(\pi j/n) x_j$$

is then transformed to the frequency domain, giving the Fourier series cosine and sine coefficients

$$a'_k = \frac{2}{n} \sum_{j=0}^{n-1} x'_j \cos(2\pi jk/n)$$

and

$$b'_k = \frac{2}{n} \sum_{j=0}^{n-1} x'_j \sin(2\pi jk/n)$$

for $k = 0,1,\ldots n/2$. Alternatively, we can compute the Fourier series coefficients a_k and b_k of the time series $\{x_j\}$, then window in the frequency domain with the operations

$$a'_k = \frac{1}{4}(-a_{k-1} + 2a_k - a_{k+1})$$

and

$$b_k = \frac{1}{4}(-b_{k-1} + 2b_k - b_{k+1})$$

for $k = 0,1,\ldots n/2$. In computing the end values, the following properties are used: a is an even function

(i.e., $a_{-k} = a_k$), b is an odd function (i.e., $b_{-k} = -b_k$), and both have period n. Both approaches give the same coefficient values a'_k and b'_k. Since we use a stored table of the sine function in the Fourier transform program, we find it slightly faster to apply the window in the time, rather than frequency, domain.

We next compute the estimated power spectrum

$$P_k = \frac{1}{2}\left[(a'_k)^2 + (b'_k)^2 \right]$$

for k = 0,1,... n/2. The values nP_k/f_s then estimate the power spectral density function with integrated power (area under the curve) equal to

$$\frac{PO}{2} + \sum_{k=1}^{\frac{n}{2}-1} P_k + \frac{1}{2} P_{\frac{n}{2}},$$

the total power of the windowed time series. With the window function of Eq. (3), the corresponding autocovariance function is

$$C_\tau = \frac{1}{n} \sum_{j=0}^{n-1} \sin^2(\pi j/n)\sin^2(\pi(j+\tau)/n)x_j x_{(j+\tau) \bmod n}$$

for τ = 0,1,... n/2. This function is similar to ψ_τ' in Eq. (2), except for the inclusion of a data-window function.

In the absence of noise we could stop at this point. With noise we smooth the power spectrum estimates, reducing the resolution in exchange for a gain in statistical stability. A convenient weighting for this purpose is

$$P'_k = \frac{1}{4}(P_{k-1} + 2P_k + P_{k+1})$$

using at the ends the property that P is an even function with period n. In the time domain this smoothing is equivalent to truncating the autocovariance function with the window

$$C'_\tau = \cos^2(\pi\tau/n)C_\tau$$

for τ = 0,1,... n/2. Although gradual, this window function effectively truncates the autocovariance function at about 50 percent of maximum lag. This is a simple case of triangular smoothing. If additional averaging is needed, the general triangular smoothings,

$$P'_k = \frac{1}{2^m} \sum_{j=-m+1}^{m-1} (m - |j|)P_j$$

for k = 0,1,... n/2 are useful, and can be computed quickly, and without multiplication except for scaling by $1/2^m$; this class of weighting function is discussed further by Singleton and Poulter.[6]

For a given sampling frequency, the resolution cell width f_s/n (i.e., the spacing of spectral estimates) is determined by the number n of sample points in an analysis. Improvement of resolution argues for large sample sets. On the other hand, if the spectrum is changing over time, spectral peaks broaden, and resolution is lost through use of a sample set spanning too long a time interval. Some balance between these factors is needed in choosing the time span for each analysis.

The techniques of digital spectral analysis are applicable to a wide range of signal types. In some cases we are interested in measuring the spectrum of a periodic or almost periodic source, with random noise regarded as a nuisance component. The effect of this noise can be reduced by averaging the power spectrum, although spectral peaks are also broadened. In other cases the source produces a random signal according to an unknown continuous power spectral density function, and our estimate of this density function is subject to random variations. For a process of this class we either use more smoothing on the result or examine instead the estimated spectral distribution function, the cumulative of the density function. Between these extremes are the mixed spectrum cases; here we may need to examine the estimated spectrum with more than one smoothing to bring out the features of interest.

The fast Fourier transform

The computational key to doing spectral analysis in real time at relatively high sampling rates is the fast Fourier transform algorithm. The algorithm we use is due to Cooley and Tukey.[7] Other authors[4,8,9] have made subsequent contributions to the development of this algorithm.

The fast Fourier transform algorithm is a method for computing the complex Fourier transform

$$\alpha_k = \frac{1}{n} \sum_{j=0}^{n-1} x_j \exp(i2\pi jk/n)$$

for k = 0,1,... n-1, where x_j are complex data values and α_k are complex Fourier coefficients. The algorithm can also be used to evaluate the inverse transform

$$x_j = \sum_{k=0}^{n-1} \bar{\alpha}_k \exp(-i2\pi jk/n)$$

for j = 0,1,... n-1. The method of computation is based on decomposing n into its factors n_j for j = 1,2,... m, where

$$n = \prod_{j=1}^{m} n_j.$$

The transform is computed in m stages, with n/n_j transforms of dimension n_j evaluated at the j^{th} stage. The number of arithmetic operations is proportional to

$$n \sum_{j=1}^{m} n_j$$

instead of n^2 as in most other methods—a substantial saving when n has more than one prime factor. Clearly, small factors are advantageous.

In the system described in this paper, n is assumed to be a power of two and we use factors of two in the transform. Factors of four or eight give some gain in efficiency, taking better advantage of multiples of $\pi/2$ and $\pi/4$, but require larger programs. This possibility should be investigated if sufficient program storage is available. Other factors, such as three or five, can be provided for if needed by added coding, without significantly changing the maximum bandwidth of the system.

Since the input data are real-valued rather than complex, we can take advantage of the complex conjugate symmetry

$$\alpha^*_{n-k} = \alpha_k$$

for k = 1,2,... n/2 of a complex transform of real data, and transform two sequences of real values at the same time, storing one as the real component and the other as the imaginary component of the complex vector to be transformed. The symmetry property allows separation of the two transforms. In a single input system, we take the even- and odd-numbered data values for the two real sequences, compute a complex transform of dimension one-half the number of data points, separate the cosine and sine coefficients for the two sequences, and then combine results as a final step to give the Fourier series cosine and sine term coefficients for the full set. In a system with dual input channels, and assuming equal sample sizes for the two channels, we simply separate the results after computing the complex transform.

When the fast Fourier transform is computed in place—that is, with the results of each elementary transform of dimension n_j stored in place of the n_j complex values used in computing the transform—the final result must be permuted to bring it into normal order. Alternatively, the data can be permuted initially and the transform computation rearranged to give the final result in normal order. The choice between the two approaches makes little difference in computing time; we have used both ways, and now permute prior to the transform, while storing the initial input data values. For n a power of two, a reverse binary permutation is required—that is, the data storage locations are indexed in inverted bit order. In our real-time system, a stored permutation table of length ¼ the transform dimension (⅛ the sample size) is used for this purpose. This table can be eliminated with some loss in computing speed if additional storage space is needed; a programmed reverse binary counter would then be used.

As is typical of many small computers, the SDS 930 computer used in our real-time analysis system performs fixed-point arithmetic considerably faster then floating-point. This led us to consider the possibility of computing the last Fourier transform with fixed-point arithmetic. When we examine the computational steps, we find that if the complex data values are scaled initially to be less than one in magnitude, and are then scaled by an additional factor $1/n_1$ prior to the first stage, where n_1 is the first factor of n in the transform, the complex values on completion of the first stage are again less than one in magnitude. If we continue, scaling prior to each stage by the reciprocal of the factor of n used in that stage, the final and all intermediate complex values remain less than one in magnitude. The overall scaling during the transform is 1/n. For our program, using only factors of two, we scale the real input data values to be less than 0.25 in magnitude, yielding initial complex values less than $\sqrt{2}/4$ in magnitude. The results of each two-by-two transform in the first phase are then less than one in magnitude (in fact, less than $\sqrt{2}/2$), and are scaled by 1/2 before storing. The intermediate results are similarly scaled by 1/2 between succeeding stages of the transform computation. The final step of separating the Fourier coefficients for the even- and odd-numbered data entries and combining the results induces a multiplication by 2, yielding a value of $|a_0| < 1.0$, and all other Fourier cosine and sine coefficients < 0.5 in magnitude. With this scaling, all numbers remain less than one in magnitude, yet good accuracy is maintained.

Bandwidth

In using the term *maximum bandwidth*, we refer to the maximum bandwidth attainable without omitting segments of the original signal. If the computing time for a given number of sample points is greater than the time span of the sample, we must sample bursts of data. In this case the spectral bandwidth may exceed the maximum for continuous sampling, but some data

is lost. The seriousness of this loss will depend on the rate of change of the spectral content of the signal.

We can, on the other hand, overlap samples, with a corresponding decrease in bandwidth. When using the data-window function (3), overlapping samples by 50 percent gives improved utilization of the available data. In this way, each sample value is divided between two sample sets, with a combined weight of one; however, the maximum bandwidth is reduced by one-half.

In a multiple-input system, the available maximum bandwidth is shared among the several channels. With two channels and samples of equal size, the bandwidth is approximately one-half that for a single channel. In this case, a single complex Fourier transform can be used to transform the two samples. With a relatively simple change, the transform program can handle any even number of equal-sized samples, with some saving in indexing time.

Dynamic range

In a digital spectral analysis a number of factors combine to give a base noise level in the frequency domain, even without noise in the original signal. In order to distinguish a periodic component in the signal, it must stand above the random fluctuations in the base noise level in the frequency domain. The main factors determining this base level are response to single tone input, the analog-to-digital quantization noise level, the sample length, and the computer roundoff noise.

First we establish a reference level with which to compare the base noise. Suppose we have an analog-to-digital converter with k-bit output, giving 2^k possible sample levels, and assume a sine function signal. Using the full range of the converter, the rms level of the signal is $2^{k-1}/\sqrt{2}$, or, in dB,

$$\text{signal level} = 20 \log_{10}(2^{k-1}/\sqrt{2}) = 10(2k - 3)\log_{10}(2).$$

If we assume the converter to be a noise source, adding a random component with uniform distribution on the interval $[-\frac{1}{2},\frac{1}{2}]$ to the signal, the resulting noise spectrum is nearly uniform over the 2^{m-1} spectral estimates for a sample of 2^m data values. The average power level (variance) of this uniform noise can be shown to be 1/12. Thus the noise level of the spectrum from this source is approximately

$$\text{noise level} = 10 \log_{10}(1/(12 \times 2^{m-1})) = -10\ m \log_{10}(2) - 10 \log_{10}(6).$$

The resulting signal-to-noise ratio of the spectrum in dB is then

$$\text{signal/noise} = 10(2k + m - 3)\log_{10}(2) + 10 \log_{10}(6) \approx 3(2k + m) - 1.25,$$

a result significantly better than the quantization range of the ADC. We see that doubling the sample size gives a 3-dB improvement in signal-to-noise ratio ($10 \log_{10}2$). Each additional ADC bit—i.e., doubling the number of possible converter output levels—gives a 6-dB improvement in signal-to-noise ratio. Experimental results give good agreement with the performance predicted by the above formula. With $k = 8$ for our converter, and with a sample of 2048 data points ($m = 11$), the predicted signal-to-noise ratio is 80 dB; this is about the average base noise level observed experimentally for a sample of this size. Use of a 14-bit converter, a commercially available product, would raise the theoretical dynamic range to 116 dB for $n = 2048$.

The power spectrum of an unwindowed signal of unit amplitude at frequency $\frac{xf_s}{n}$ is approximately

$$\frac{1}{2}\left[\frac{\sin(\pi(k-x))}{n \sin(\pi(k-x)/n)}\right]^2 \tag{4}$$

for values of k near x. If x is an integer, the power-spectrum value is 1/2 at $j = x$, and zero elsewhere. In general, however, the power spectrum of an unwindowed signal falls off slowly with $|x-j|$, and the dynamic range is seriously limited. Using the data window in Eq. (3), the main peak of the response function is broadened, but beyond two resolution cells falls off more rapidly with $|x-j|$ than function (4), fast enough to be below the usual analog-to-digital converter noise level by 10 resolution cells away from a periodic signal at the maximum amplitude allowed by the converter. In most cases this is adequate data windowing to eliminate response shape (leakage) as a limiting factor in dynamic range.

With typical computer word lengths, computer roundoff noise is negligible compared with the other two factors we have discussed. Since the fast Fourier transform of $n = 2^m$ real data points requires only m steps, including the final step to separate and recombine the transform of the even- and odd-numbered data values, round-off error in the transform is low, even when using fixed-point arithmetic. However, in squaring the sine and cosine coefficients to form the power spectrum, double precision results should be retained or a shift to floating-point arithmetic made. Otherwise, the potential dynamic range of the computations will be reduced. The computer in our real-time analysis system has 24-bit words, including sign, which is more than adequate in comparison with the 8-bit analog-to-digital converter. On the other hand, a 12-bit com-

puter word-length would result in computer round-off being a more important factor in limiting dynamic range than the analog-to-digital converter.

The ability to distinguish two tones as separate peaks in the power spectrum depends mainly on their relative levels and the number of resolution cells separation between the two tones. For tones of equal amplitude, a minimum separation of three resolution cells (i.e., $3f_s/n$), is needed to give a dip between peaks. For detection of a weak peak near a strong one, as in Doppler signal analysis, more separation is needed. And the weak peak must, of course, be above the base noise level of the spectrum. In Figure 2 we show the results of an experiment to study the effect of windowing in two-tone resolution. These analyses were computed with n = 512, giving 256 resolution cells. Smooth curves were drawn through the discrete points. The simulated signal was

$$\sin(\pi j\ 128.5/256) + 0.001\ \sin(\pi j\ 137/256)$$

for j = 0 1,... 511. The first component is midway between two multiples of f_s/n, and the second is 60 dB down and 8.5 resolution cells higher in frequency. The power spectrum was computed without data windowing, with the data window

$$\sin^2(\pi j/n) \qquad (5)$$

and with a second data window

$$\sin^4(\pi j/n). \qquad (6)$$

The signal 60 dB down and displaced by 8.5 resolution cells can be seen in the runs with window function (5) and (6), but not in the unwindowed run.

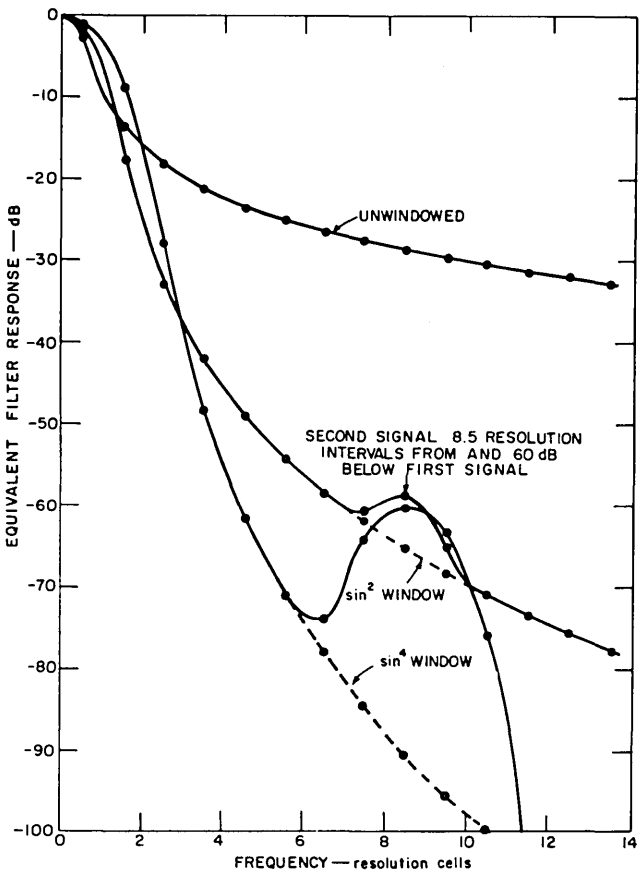


Figure 2 – Effect of windowing on two-tone dynamic range

Implementation of a real-time digital analysis system

The components of a typical real-time digital spectral analysis system are shown in Figure 3. This diagram illustrates a multiple-analysis-channel system, designed for simultaneous processing of several spectral channels. A high-speed digital switch (or multiplexer) under computer control is used to sequence among the analog input channels. An ADC is used to convert the voltage level on the designated channel to a binary-coded sample value. Once the power spectrum is calculated, the values may be displayed or used as data for further calculations. With most display units, some form of digital-to-analog converter (DAC) is needed. Digital interface circuitry linking the ADC and DAC units to the central computer is also needed.

The necessary size of high-speed memory for a computer in a real-time analysis system is mainly a function of the maximum number n of data points to be processed in an analysis interval-i.e., of the sum of the sample sizes for each of the input channels in a multi-channel system. The minimum storage required is of the order of 2n for data, n locations for the data currently being transformed, another n locations for storing the next data set, and several hundred locations for program storage. Adding table storage of the sine and reverse binary functions, we arrive at a practical minimum of 3n locations for data, program, and tables. In this we make no allowance for separate output buffer areas. We assume either that the output is completed by high-speed transfer prior to the start of computing for the next sample, or that a common input-output buffer area is used. In the latter case, an output operation is interleaved between each pair of input operations; every second input value replaces an output value that has already been used, and the output is clocked at half the input rate. If equipment limitations require separate input and output buffer areas, then one or two additional areas of size n/2 will be needed for output.

In selecting a computer for real-time spectral analysis, input-output buffering independent of the main program is a desirable feature to have. Lacking this, an efficient method of external interrupt is essential

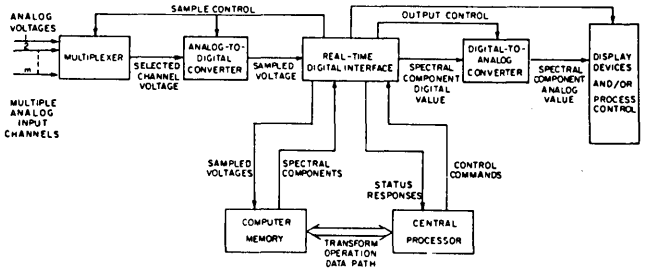


Figure 3 – Components of a real-time spectral analysis system

in processing real-time data. If data must be moved between buffer areas and working storage arrays, the capability to rapidly transfer groups of words is also important. However in our demonstration system input and working storage arrays interchange roles for alternate data sets, thus avoiding the need for actual transfer of location.

In the fast Fourier transform for n a power of two, the inner loop of the algorithm performs a computation on a pair (x_j,x_k) of complex data values, and stores the results (y_j,y_k) in the original locations. This loop accounts for over 50 percent of the total computing time. The complex arithmetic operations are either

$$y_j = x_j + x_k e^{i\theta}$$

and

$$y_k = x_j - x_k e^{i\theta}$$

or

$$y_j = x_j + x_k$$

and

$$y_k = (x_j - x_k)e^{i\theta}$$

depending on the particular form of the algorithm used. The two forms require an equal number of arithmetic operations. When expanded into real and imaginary parts, Re(·) and Im(·), the second form is

$$\mathrm{Re}(y_j) = \mathrm{Re}(x_j) + \mathrm{Re}(x_k)$$
$$\mathrm{Im}(y_j) = \mathrm{Im}(x_j) + \mathrm{Im}(x_k)$$

$$[\mathrm{Re}(x_j) = [\mathrm{Re}(x_j) - \mathrm{Re}(x_k)]\cos\theta - [\mathrm{Im}(x_j) - \mathrm{Im}(x_k]\sin\theta$$
$$[\mathrm{Re}(x_j) = [\mathrm{Re}(x_j) - \mathrm{Re}(x_k)]\sin\theta + [\mathrm{Im}(x_j) - \mathrm{Em}(x_k)]\sin\theta$$

Since complex values require two storage locations, four array entries are used in the calculation. The locations of the four entries are incremented by a common constant each time through the loop, until a limit is reached. This constant, the separation k-j, and the angle θ are varied outside this loop. In comparing computers for possible use in real-time spectral analysis, the estimated times for this loop should give a good relative measure of computing speeds. Time for this loop on our SDS 930 computer is 187 μs. Multiple arithmetic registers can be used to advantage in these computations; efficient address modification is also a valuable asset for a computer in this application.

Description of SRI's real-time spectral analysis system

An experimental real-time spectral analysis system using a general-purpose digital computer has been assembled and tested at Stanford Research Institute. A relatively small version of the SDS 930 digital computer is used. This computer has 8192 24-bit memory locations, which limits the sample size to 2048 values. It has one index register. Addition time is 3.5 μs and multiply time is 7.0 μs. To reduce computing time, the program for real-time spectral analysis is written in machine language, and sine function and reverse-binary tables are used. A sample of 1024 data points can be transformed in just under one second. Times for an earlier FORTRAN version of the program, using floating-point arithmetic, were slower by a factor of ten.

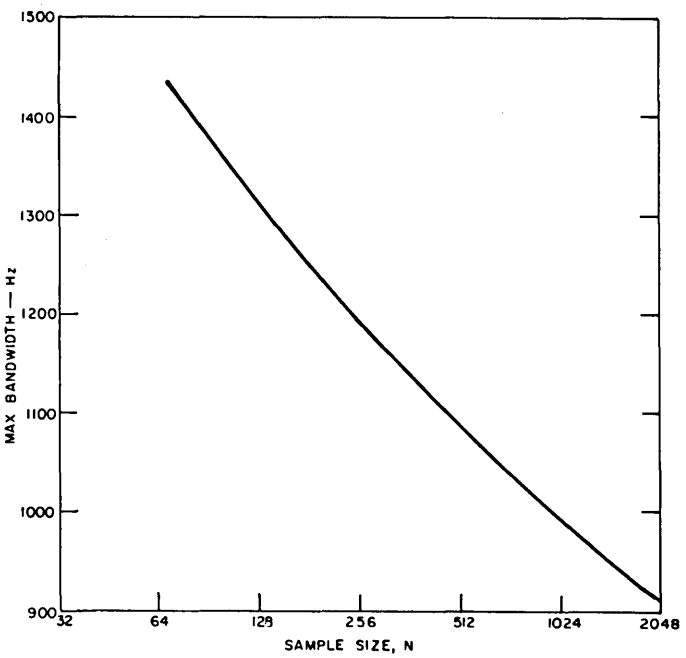


Figure 4 – Measured maximum bandwidth for real-time analysis system

Input to the system is through an 8-bit multiplexed ADC; output leaves via two parallel 12-bit DACs, under computer control through a digital interface. Input of sample values is controlled by real-time interrupts. The input-output interface will permit only single word operations; thus the main program is interrupted for each input or output transfer. Maximum bandwidth for single-channel operation is approximately 1000 Hz, but varies somewhat with sample size as shown in Figure 4. At the maximum sample size of 2048 values, the measured dynamic range is approximately the 80-dB level predicted theoretically.

This digital spectrum analysis system has been used to provide spectral measurements in a variety of research areas. The input data for these measurements has typically been recorded on analog magnetic tape. This is unfortunate in that the potential dynamic range of the measurement is limited by the characteristics of the analog record-reproduce equipment; the full dynamic range of the digital system is not used. However, the resulting range is considerably better than with currently available analog analysis equipment. In addition, the real-time speed of analysis allows tape playback at the same speed as recorded. Equalization problems are minimized.

The analysis program allows two forms of output for the estimated power spectrum. In one option, the results are recorded on magnetic tape for later use. In the other option, spectral estimates are sequentially read out through a DAC in step with a ramp-function (time sweep) on a second DAC to produce a two-axis display. The results may be observed on an oscilloscope, and photographed, or used to drive an intensity-modulated electrographic chart recorder. In the latter case, the power spectrum is logarithmically scaled to better display the dynamic range of the measurements. Although these devices work reasonably well, we feel there is a need for better display equipment for real-time spectral analysis—i.e., equipment with improved resolution, dynamic range, and ability to operate at real-time speeds with a minimum of computer formatting and control.

The principal use we have made of this system is in the analysis of Doppler radar signals. The high analysis dynamic range has also made this spectral measurement technique valuable for the on-line examination of frequency synthesizer output for spurious content and for measurement of radar receiver frequency response. This initial system has served as a useful tool in the development of theory and practice for real-time digital spectral analysis.

Possibilities for bandwidth improvement

The main limitation of the present system is bandwidth. The dynamic range is excellent, and can be improved by buying an ADC with a greater number of output bits, at a small increase in total system cost. The resolution is also excellent, and can be increased if necessary by providing additional high-speed storage for data. Bandwidth, however, can be increased only a small amount by means other than substituting another, faster computer. Addition of an independent input-output buffering system, not interrupting the main program, would increase bandwidth in the order of 10 percent.

Multiple arithmetic units and ultra-high-speed "scratch pad" memory can be used to good advantage in computing the basic two-by-two transform loop. The Control Data 6800, one of the several large computers with these features, would, for example, allow a system bandwidth in excess of 70 kHz. Another possibility is to add special-purpose hardware for complex arithmetic, or for computing the entire two-by-two transform, to a general-purpose computer.

By further use of parallel processing, system bandwidths of the order of one megahertz or larger would be possible. The fast Fourier transform is an ideal candidate for parallel processing, since for each factor n_j of n we compute n/n_j independent transforms of dimension n_j. Thus we could use n/n_j processors operating in parallel. A multiple parallel processor for the fast Fourier transform could be constructed as a special-purpose device, using integrated circuit techniques. The wiring-plan simplicity of one particular organization of the fast transform for computing with these techniques has been noted by Singleton[9] and by Pease and Goldberg.[10] With this organization, a transform of dimension 2^m is computed in m steps, while copying back and forth between two data vectors with fixed interconnection paths. A preliminary study by Pease and Goldberg[10] indicates that system bandwidth can be increased by a factor of 1000 or more through parallel computing. Bergland and Hale[11] have proposed another organization, using m parallel arithmetic units for a transform of dimension 2^m, but with serial computing within each of the m steps. This proposal also offers promise of greater improved bandwidth. In either case, the parallel processor could be an auxiliary device connected to a general-purpose computer.

CONCLUSION

The fast Fourier transform algorithm has led to revision in the methods of spectral analysis. It has also reduced computing time for spectral-analysis calculations to a level where current-generation general-purpose computers can perform spectral analysis at real-time rates and with bandwidths that are sufficient for many problems. Advantages of high resolution, high dynamic range, repeatability, and of flexibility of method through programmed, rather than competitive with analog spectral analyzers.

We have assembled a real-time spectral analysis system, using an available, small general-purpose digital computer. The experimental results on this system have met theoretical expectations. In this system we have demonstrated a powerful new tool

for real-time analysis of non-stationary time functions in the presence of noise.

Future developments in digital technology will lead not only to faster and more versatile small general-purpose computers, but probably also to parallel Fourier transform processors as adjuncts to these computers. The greatly increased bandwidth can be expected to bring many more spectral analysis applications into the realm of real-time digital processing.

ACKNOWLEDGMENTS

The authors wish to thank the many Stanford Research Institute staff members who gave helpful suggestions and encouragement in this work.

REFERENCES

1 R B BLACKMAN J W TUKEY
The measurement of power spectra
Dover Publications Inc New York 1958

2 C BINGHAM M D GODFREY J W TUKEY
Modern techniques of power spectral estimation
IEEE Trans. on Audio and Electroacoustics Vol AU-15 No 2 pp 56-66 June 1967 This is a special issue on the last Fourier transform and its application to digital filtering and spectral analysis

3 T G STOCKHAM
High-speed convolution and correlation
AFIPS Conference Proceedings 1966 SJCC Vol 28 pp 229-233 Spartan Books Washington D C 1966

4 W M GENTLEMAN G SANDE
Fast Fourier transforms—for fun and profit
AFIPS Conference Proceedings 1966 FJCC Vol 29 pp 563-578 Spartan Books Washington D C 1966

5 R C SINGLETON
An ALGOL convolution procedure based on the fast Fourier transform
SRI Project 181531-132 Stanford Research Institute Menlo Park Calif January 1967 AD-646 628

6 R C SINGLETON T C POULTER
Spectral analysis of the call of the male killer whale
IEEE Trans on audio and electroacoustics Vol AU-15 No 2 pp 104-113 June 1967

7 J W COOLEY J W TUKEY
An algorithm for the machine calculation of complex Fourier series
Mathematics of computation Vol 19 No 90 pp 297-301 April 1965

8 R C SINGLETON
On computing the fast Fourier transform
Communications of the ACM Vol 10 No 10 *October 1967*

9 R C SINGLETON
A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage
IEEE trans on audio and electroacoustics Vol AU-15 No 2 pp 91-98 June 1967

10 M C PEASE J GOLDBERG
Investigation of a special-purpose digital computer for on-line Fourier analysis
Special technical report No 1 project 6557 Stanford Research Institute Menlo Park Ca ifornia April 1967

11 G D BERGLAND H W HALE
Digital real-time spectral analysis
IEEE Trans on electronic computers Vol EC-16 No 2 pp 180-185 April 1967

Further advances in two-dimensional input-output by typewriter terminals**

by MELVIN KLERER* and FRED GROSSMAN
Columbia University, Hudson Laboratories
Dobbs Ferry, New York

INTRODUCTION

The purpose of this paper is to describe some new applications in the realm of two-dimensional input-output by typewriter terminal. These applications are extensions of the software and hardware system techniques covered in the references 1-4, 11-21. The principal elements of this approach consist of a versatile programming language and a reverse-indexing typewriter equipped with a special character set. This set permits not only normal alphanumerics but also the construction of arbitrarily-sized symbols by the use of a few interlocking primitive strokes. The reverse-indexing and indexing, together with spacing and backspacing, allow full two-dimensional keyboard control of the typed document. Typing errors are corrected by moving to the desired point and overtyping, or by pressing an "erase" button.

We have worked with computer input-output typewriter terminals primarily because of their ubiquitous nature. Thus we have sought to demonstrate that a flexible computer language and input-output capability could be made available with essentially conventional equipment. Nonetheless, where alternatives were possible, we have chosen those directions which could be easily implemented on more flexible (and more expensive) devices. Specifically, all of the examples cited in this paper could be implemented, using exactly the same software techniques, on keyboard controlled cathode ray tube displays designed for retentivity of alphanumerical information. The use of red alphanumerics, easily available on a typewriter, could be effected in the black and white tube by dotted or dashed characters (or a different type style). The major limitation of the CRT is its relatively small visual working area; but a 20-inch carriage typewriter, similar to our own, could be used as an accessory device. Essentially the same output techniques could drive a conventional paper-tape controlled typesetter, photo-composition device, or a high speed printer with a special character set. If extended subscripting and superscripting (indexing and reverse-indexing) are available as a hardware facility, then no modification of the output methods is necessary. If subscripting/superscripting is not available, then the order of output can be modified to move left to right, line by line. The final output would be the same. The only difference is that the latter method takes more time since unnecessary blank space must be scanned. Also it is more critically dependent on the alignment of the machine when interlocking primitive strokes are used to construct large two-dimensional characters.[12,13,16] Therefore, further development and extension of these techniques with a high speed photo-composition device can be used to produce or edit documented programs, mathematical text, or text with displayed mathematical equations to a high degree of typographic quality, a good deal of this process being automatic.

*A two-dimensional format statement**

The purpose of this language facility is to allow two degrees of freedom for formatted printing of computed labels. These are embedded in a two-dimensional construction. In order to keep within the user oriented framework established in earlier work, we felt that the output should be a literal image of the input. Further, the relative location, numerical size, and numerical character of computed labels should be indicated by simply "marking" the desired location rather than by counting spaces and inputting coordinate information. The *com-*

* Present address: Department of Industrial Engineering, New York University, New York, N.Y. 10453
**This work was supported by the Office of Naval Research and the Advanced Research Projects Agency.

* An example of this statement was demonstrated at the 1965 FJCC in connection with the verbal presentation of the paper cited in reference 16. However, it was not documented in that paper or subsequent published work.

mand to compute the required labels is exemplified by the statement *PRINT IMAGE N, X, Y, E, F, Z = G*. *N* is the numerical identifier for the *IMAGE* statement. *X* and *Y* are variables which have been previously computed. *E* and *F* stand for arithmetic expressions to be computed which themselves may contain two-dimensional constructions. That is, the expressions *E* and *F* may contain superscripted or subscripted quantities, the numerator over denominator form of division and the integral, sum, product or square root operators displayed as arbitrarily sized symbols. The artifice $Z = G$ indicates that the expression *G* is not only to be computed and printed but also should be stored as the value of the variable *Z*.

An example of a format definition is

```
     LL      LLLLLLL xxx.xx LLL LLL
       L
        LL
IMAGE N      LLLLLLL xxxxxx.xx LLL   L LLL
                                  xx
                                  LLLL    L
          LLLLLLL xxxx LL LLLy
```

where the *L*'s stand for any typable character or any constructable symbol. A symbol, such as a constructed arbitrarily sized sigma is interpreted as a summation operator if it appears in the *PRINT* (image) command. But if it appears in the *IMAGE* statement, which *is* the format, it will be interpreted as a *picture,* not as an operator. The picture linked with the summation operator could just as well be the drawing of a house. The *values* of X, Y, E, F and G are the *computed labels* inserted where the lower case *x*'s appear in the *IMAGE* format statement. The number of lower case *x*'s in each set indicates the maximum (rounded) integer size or the maximum number of digits to the left or right of the decimal point. A lower case *y* indicates that a computed label in floating point notation is to be inserted at the indicated location. The linkage between each set of lower case *x*'s or lower *y*'s and the order of the computed labels (reading from left-to-right) is governed by a precedence rule. The first computed label is inserted into the location indicated by that set of lower case *x*'s (or lower case *y*) that are the left-most and highest. The second computed label is inserted into the location designated by that set of lower case *x*'s (or lower case *y*) that is the next highest at that columnar position or next left-most.

The first line of Figure 1 illustrates a print image statement. An integral, a sum, and a square root are computed for various values of the parameter *i*. The *SLEW 5* phrase indicates that five blank lines are to be inserted between each image. The several values of *i* that appear in the *PRINT* image command are inserted for format control so that the specific value of *i* used for each image can be indicated in the pictorial output. The second line of Figure 1 is the two-dimensional format statement and indicates where the computed labels are to be placed. The third line of Figure 1 is the *output* of the program for the parameter $i = 2$. The fourth line is the output for $i = 3$, and so on. The upper part of Figure 2 represents an edited version of this program when operating in an on-line, interactive mode. The original *PRINT* image statement has not been altered. However, the *IMAGE 1* format has been replaced by another. A *drawing* of a "house" has been substituted for the integral symbol, a box has been circumscribed about the summation symbol, and the square root symbol has been replaced by another indiosyncratic version. Also the last computed label has been changed to fixed point (*x*'s), rather than floating point (*y*). The lower four lines of Figure 2 represent the *output* of the new *IMAGE* for the values $i = 2, 3, 4, 5$ respectively.

Figure 3 represents the input *and* output of a program run in an on-line interactive mode. The program input in the upper part of the figure is terminated by the word *FINISH*. The word *INPUT* is typed by the system indicating that it is ready to accept input data. The numbers that follow, *12, 2.5, 10, 3.5, 6* are the first set of values for V, R_1, R_2, R_3, and R_4 respectively. The lower part of Figure 3 is the program *output* where computed labels have been inserted into the two-dimensional figure. The program will then request a new set of input data and output another figure with new values.

Editing two-dimensional programs or text

One way of editing programs is by inserting or deleting entire statements or statement segments under control of such editing instructions as *EDIT, INSERT, DELETE, OMIT, LIST,* etc. For a programming language that is restricted to short statements, such an approach might be completely satisfactory. However, although such commands are permitted in the basic system,[12,13,16] they do not prove to be as satisfactory as they would be for the more restrictive languages. This is because the basic language allows quite long statements that may extend over several lines (a maximum 18-inch typed line is possible on our Flexowriter model 13 inches on our Selectric model) and permits complex two-dimensional constructions. Therefore the need was felt for an editing technique that would allow character editing as easily as statement editing while still keeping full two-dimensional control for character or constructed symbol insertion. In keeping with the user oriented approach of the original programming system, the technique was designed to minimize the number of necessary formal conventions.

A first verison of this special editing program is now operable in an off-line input mode. Therefore all subsequent remarks in this section will refer to a procedure

FROM 1=2 TO 5 SLEW 5, PRINT IMAGE 1,1,1,1, $A_1=\int_0^1 \frac{e^{-1z}\ \mathrm{SINH}^{-1}\frac{z}{2}}{z^5+\frac{1}{2}}dz$, 1,1,1, $B_1=\sum_{r=1}^{1} r^1$, 1,k=1^2+1, $\sqrt{k}$.

IMAGE 1 $A_x=\int_0^x \frac{e^{-xz}\ \mathrm{SINH}^{-1}\frac{z}{2}}{z^5+\frac{1}{2}}dz$ = .xxxxx $B_x=\sum_{1=1}^{x} 1^x$=xxxx $C_x=\sqrt{xx}$=y. FINISH.

$A_2=\int_0^2 \frac{e^{-2z}\ \mathrm{SINH}^{-1}\frac{z}{2}}{z^5+\frac{1}{2}}dz$ = .12922 $B_2=\sum_{1=1}^{2} 1^2$= 5 $C_2=\sqrt{5}$ = .223606798 1

$A_3=\int_0^3 \frac{e^{-3z}\ \mathrm{SINH}^{-1}\frac{z}{2}}{z^5+\frac{1}{2}}dz$ = .07884 $B_3=\sum_{1=1}^{3} 1^3$= 36 $C_3=\sqrt{10}$ = .316227766 1

$A_4=\int_0^4 \frac{e^{-4z}\ \mathrm{SINH}^{-1}\frac{z}{2}}{z^5+\frac{1}{2}}dz$ = .05161 $B_4=\sum_{1=1}^{4} 1^4$= 354 $C_4=\sqrt{17}$ = .412310562 1

$A_5=\int_0^5 \frac{e^{-5z}\ \mathrm{SINH}^{-1}\frac{z}{2}}{z^5+\frac{1}{2}}dz$ = .03577 $B_5=\sum_{1=1}^{5} 1^5$=4425 $C_5=\sqrt{26}$ = .509901951 1

Figure 1

where the original program (or any textual material) has been typed and removed from the terminal to be proofread or compiled. The editing begins when the original *document* (the typed program) is re-inserted into the terminal and the editing *marks* and new text are typed. The typing process also produces a paper tape. When the editing operation is finished, the paper tape representing the original document is read into the system, followed by the paper tape representing the edit operation. The *output* of the system is a paper tape representing the edited program or text or, at the user's option, output may be directly on-line to the typewriter terminal. The paper tape may be reproduced as a document on the typewriter terminal or compiled as a program. The same technique, in principle, can be extended to an on-line, *interactive* mode. We have not yet done so purely for practical reasons, such as a relatively small memory in the available computer.

After the user inserts the original document (program or text) into the typewriter, he moves the platen by keyboard control to the first typed character of the document. He types a *red* underline which serves as a reference point for any subsequent scanning. If he wishes to insert a new character, word, phrase, or statement, he simply types them *just where he wants them*—if there is blank space available. To change any character(s), he can overtype with the desired character(s). Should he wish to delete any character(s), he positions the platen over the character and either presses a special "erase" button or overtypes with a red lower case or

upper case *x*. To delete an entire statement he need only type a red slash anywhere in the statement. The typing sequence for any of these actions is arbitrary.

However, there are many programs or texts that do not offer enough blank space in which to type the new characters(s), phrases, or statements desired. But there may be room in the margin, at some other place in the program, and certainly, always at the extensible "bottom" of the document.

Thus, to indicate the exact location of an insertion, the user types a red vertical. The basic character set has been designed to that the vertical stroke always falls between any two contiguous alphanumerics, so there is no possibility of an ambiguous location. Regardless of how the rest of the red *insert mark* is extended, vertically or horizontally, the position of the first red vertical determines where the left-top-most new character will be inserted. The right-most point of the new insert (relative to the left-top-most character) is determined by a red slash which also acts as the terminator of the

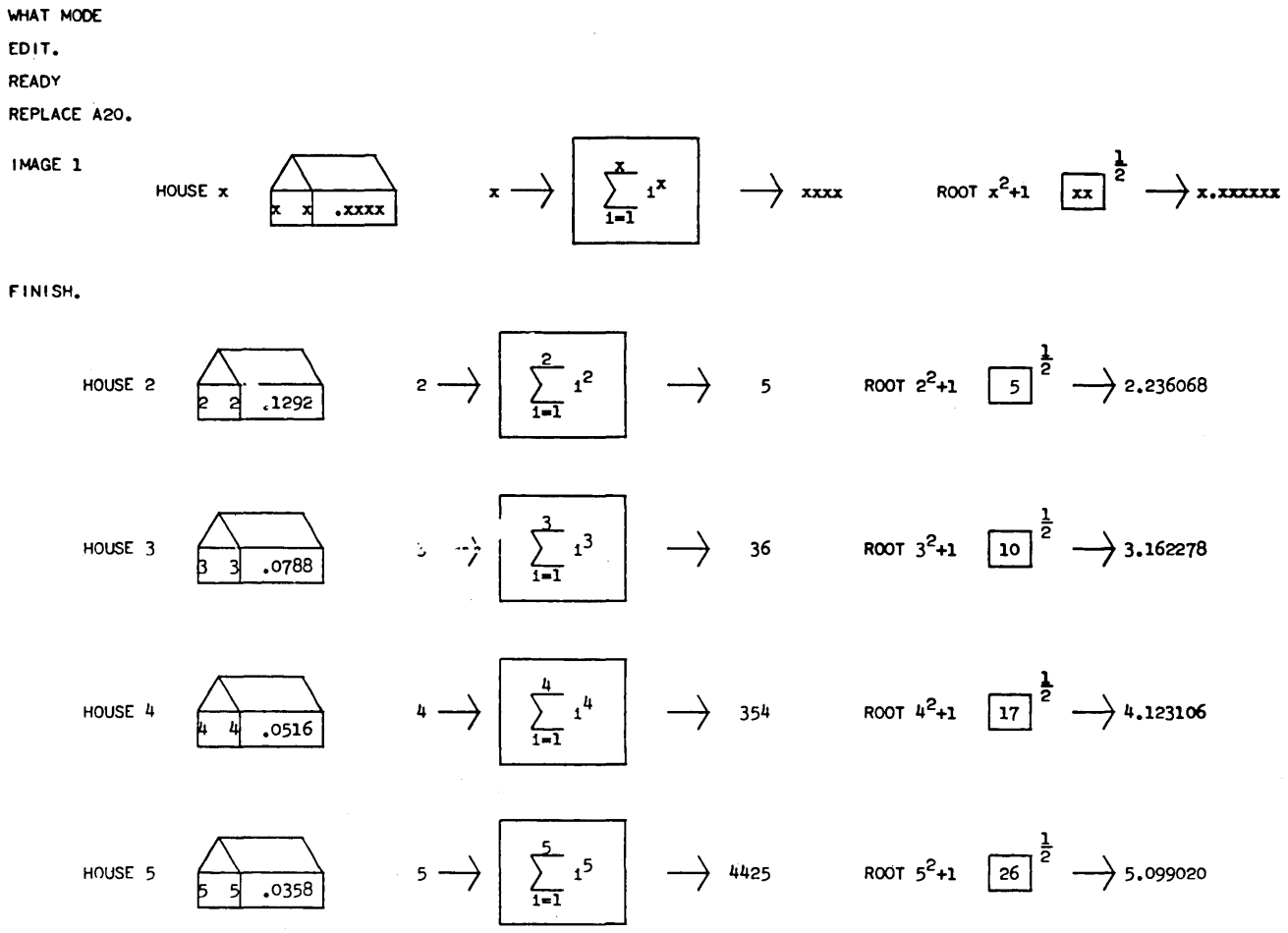
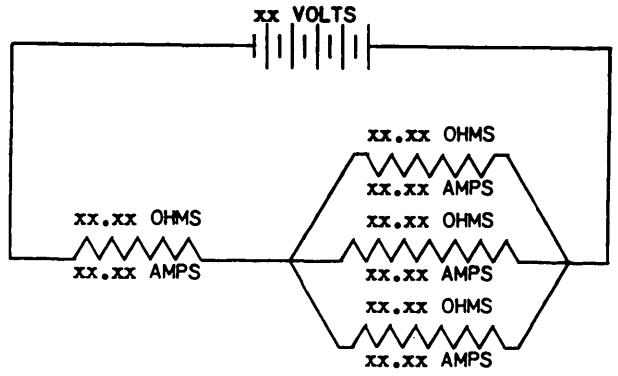


Figure 2

IMAGE 1



STATEMENT 1. READ V, R_1, R_2, R_3, R_4. $R = \frac{1}{\frac{1}{R_2} + \frac{1}{R_3} + \frac{1}{R_4}}$. $r = R + R_1$.

$I_1 = \frac{V}{r}$. $v = R \times I_1$. $I_2 = v/R_2$, $I_3 = v/R_3$, $I_4 = v/R_4$.

PRINT IMAGE 1, R_1, I_1, V, R_2, I_2, R_3, I_3, R_4, I_4 AND SLEW 10.
GO TO STATEMENT 1. FINISH.

INPUT 12 2.5 10 3.5 6

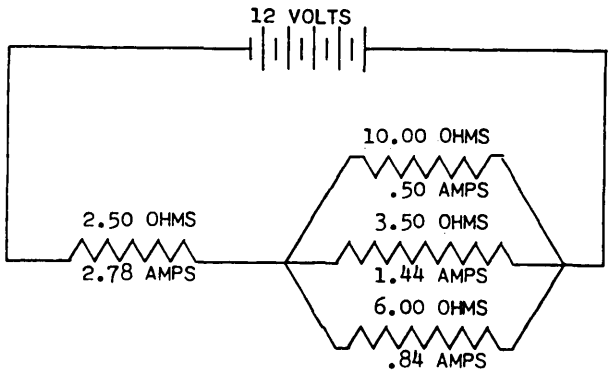


Figure 3

insert operation. Thus an insert construction consists of any figure constructed with an extensible red vertical, an *optional* extensible red horizontal and a terminating red slash. In order to increase flexibility and ease of use, leading blank spaces of the insert are ignored, but trailing blanks are entered into the edited program or text. This permits bringing the new character(s) into the margin but still giving control over space adjustments at the point of insert. Should the user change his mind after starting an insert, he can "erase" the current process by overtyping the first red vertical with a red slash. Should he change his mind while in the process of typing a new statement in a previously blank area he may cancel this by typing a red slash anywhere within the current typing. Should he wish to type the new character(s) elsewhere in the program or text, he may type *INSERT N* or just *N* within the insert mark. *N* is any red number from 0 to 9999. The actual insert may then be typed anywhere on the page as long as it is preceded by the declared insert number. Of course, any specific insert may be referenced as many times as desired.

To delete a two-dimensional *area*, two extended red vertical lines are constructed. The deleted area is defined as being bewteen the two verticals. A red slash in this area initiates deletion. (Strictly speaking, the lines need not be equal. The deleted area is defined by ignoring the extension of one line higher or lower than the other.)

For a lengthy program, there is always the possibility

of the paper getting out of alignment after it has moved large distances from the initial red underscore. To prevent this possibility, the user is advised to occasionally type over *any* black letter in the text with its red equivalent, except for the letters *X* or *F*. This will enable the system to make automatic realignment adjustments. The red letter *X*, lower case or upper case, is reserved for erasure and red *F* is used to terminate the edit operation.

Some of the possible editing techniques are illustrated in Figures 4A, 4B, 4C, and Figures 5A, 5B, 5C. In each, (A) represents the original program, (B) the program with the edit marks and new characters and (C) represents the final program output of the system.

In Figure 4B, at (1) and (4), new statements are typed directly into the desired locations. Insert marks are used at (2) and (3) where there is no available space to type directly. The actual insert designated by insert 5 at point (3) is typed (point 5) at the bottom of the sheet. At point (6) a statement is deleted by typing a red slash. Characters are inserted by typing where there is room (7) or by using any of the insert marks shown. The insert mark may be freely extended horizontally (8) or vertically (9) and the actual insert may be typed anywhere above the horizontal line, leading spaces being ignored. The terminating red slash need not come at the end of the horizontal line (8), (2), therefore not restricting the user to a neat or sequenced construction. Further, the mark may point in any direction (9). At point (10) characters are deleted by "x-ing" out and at point (11) the area to be deleted is enclosed between extended vertical lines. Note that at point (12), the "FINISH." statement has been deleted. Since the language system requires a "FINISH." statement to complete the compilation pro-

FUNCTION G(x)= $\left|\frac{\sin\frac{\pi x}{A}}{\frac{\pi x}{A}}\right|$. SLEW TOP. $M_k=\int_{-\pi}^{\pi} G(ky)dy$ FROM k=0 TO N. $C_j=\sum_{k=1}^{j} M_k$ FOR j=0(1)N .

FROM n=-1 BY .2 TO 20 AND r=0 TO 4, $B=\frac{1}{2}2^r$, $A=10^n$, N= INTEGER PART (10A),

$S=\sum_{u=0}^{N} F(u)G(u)\sin\left|G(u)\frac{2\pi Bu}{A}\right| + \sum_{k=1}^{u}(F(k)-G(k)$, PRINT n,A,S. FINISH.

Figure 4A

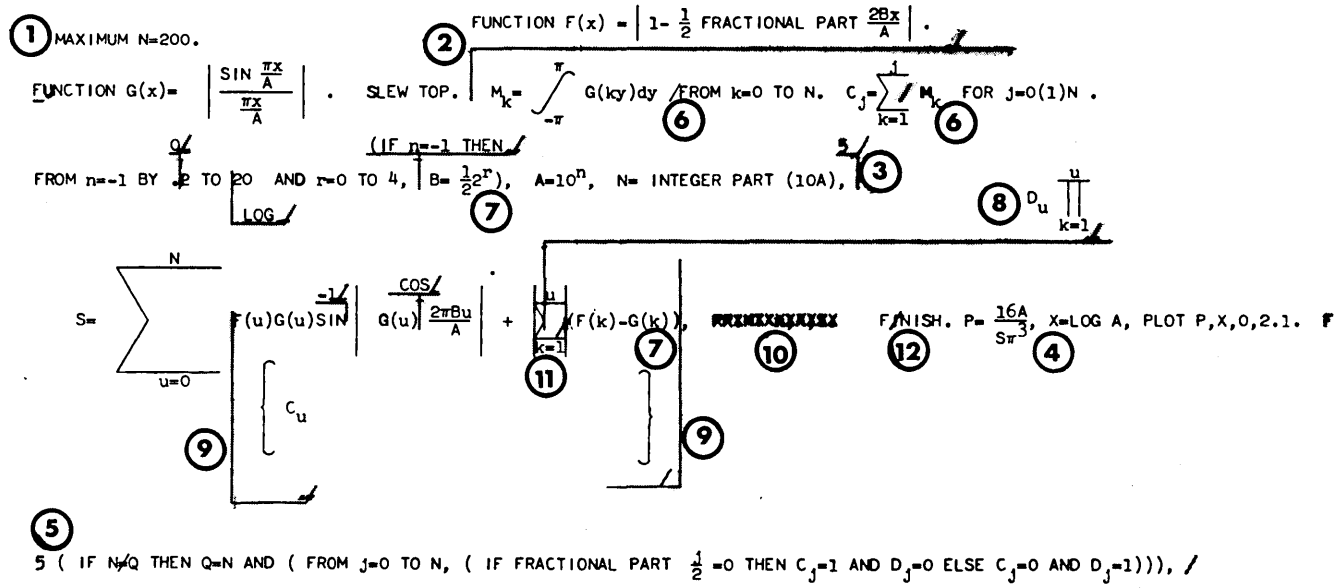


Figure 4B

cess, the edit program will supply such a statement if the user forgets (Figure 4C).

In Figure 5B, point (1) shows that inserted characters are referenced by typing *INSERT 24* and at point (2), just the number *10* is typed inside the edit mark. The actual inserts (points 3) may be typed anywhere on the page. Insert *10* has been referenced twice (2). The statement fragment at point (4) has been deleted by typing a red slash and was retyped at point (5). This statement was later re-edited at (6) where as many blank spaces as there are between the vertical and the slash of the insert mark are inserted into the final statement (as shown in Figure 5C).

In keeping with our general philosophy for educating programmers and intermittent users, we have always tried to produce "one-sheet" instruction manuals. Figure 9 is the one-sheet instruction manual that we are presently distributing to our in-house users of this editing technique. As with our previous manuals, the intent has been to eliminate the thick, and often-times incomprehensible, text supplied with many conventional systems. Hopefully, as we extend our work more into the realm

MAXIMUM N=200.

FUNCTION $G(x)= \left| \frac{\text{SIN } \frac{\pi x}{A}}{\frac{\pi x}{A}} \right|$.

SLEW TOP.

FUNCTION $F(x) = \left| 1- \frac{1}{2} \text{ FRACTIONAL PART } \frac{2Bx}{A} \right|$.

FROM $n=-1$ BY .02 TO LOG 20 AND $r=0$ TO 4, (IF $n=-1$ THEN $B= \frac{1}{2}2^r$), $A=10^n$, N= INTEGER PART (10A),

(IF $N \neq Q$ THEN $Q=N$ AND (FROM $j=0$ TO N, (IF FRACTIONAL PART $\frac{j}{2} =0$ THEN $C_j=1$ AND $D_j=0$ ELSE $C_j=0$ AND $D_j=1$))),

$$S= \sum_{u=0}^{N} \left\{ C_u F(u)G(u)\text{SIN}^{-1} \left| G(u)\cos \frac{2\pi Bu}{A} \right| + D_u \prod_{k=1}^{u} (F(k)-G(k)) \right\},$$

$P= \frac{16A}{S\pi^3}$, X=LOG A, PLOT P,X,0,2.1.

FINISH.

Figure 4C

SLEW TOP. FUNCTION $X(\varphi)= \frac{\varphi\pi}{180}$. FROM $\theta=0$ UNTIL $\text{SIN } \frac{X(\theta)}{2} < \text{TAN } \frac{1 + X(\theta)}{1 - X(\theta)}$

$$Z= \int_0^{X(\theta)} e^{-\alpha uv} \cos \left[\text{LN } \frac{\text{SIN } u + \text{TAN } u}{\text{SIN } v - \text{TAN } v} \right] dudv \text{ AND PRINT } \theta, Z.$$

FINISH.

Figure 5A

of truly interactive (man-machine) systems, even this short list of instructions will be replaced by immediate system responses to indicate if the user's communication has been correctly understood.

Automatically edited mathematical text

The original programming system was designed to recognize mathematical expressions, even when they are not typed according to a restricted syntax or in a neat format. These basic techniques may be extended to manipulate and format mathematical text. One of us (M.K.) is directing an effort to produce a table of integrals whose accuracy has been verified by various computer strategies. The algorithms used for the verification

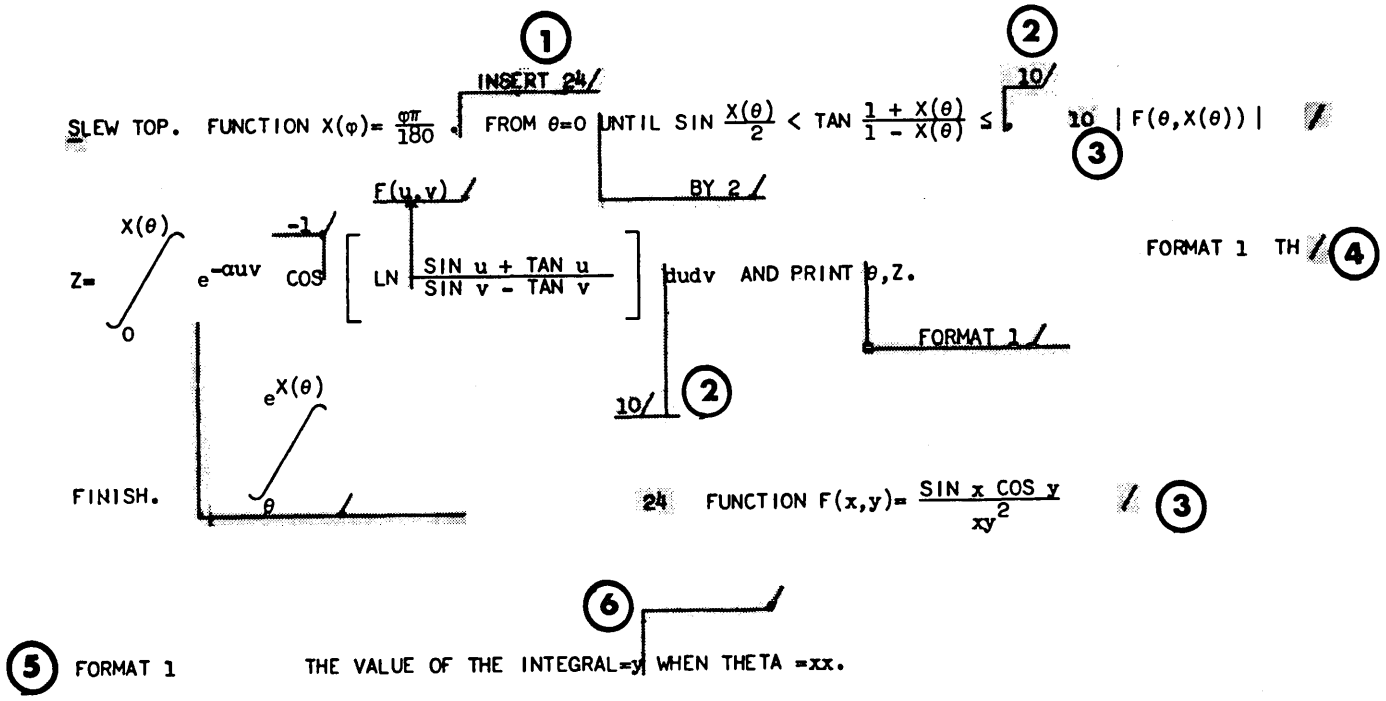


Figure 5B

```
SLEW TOP.
FUNCTION X(φ)= φπ/180 .
FUNCTION F(x,y)= SIN x COS y / xy^2
FROM θ=0 BY 2 UNTIL SIN X(θ)/2 < TAN (1 + X(θ))/(1 - X(θ)) ≤ | F(θ,X(θ)) |
Z= ∫ from 0 to X(θ) ∫ from θ to e^X(θ) e^-αuv COS^-1 [ LN F(u,v) (SIN u + TAN u)/(SIN v - TAN v) ] | F(θ,X(θ)) | dudv AND PRINT FORMAT 1 θ,Z.
FORMAT 1
THE VALUE OF THE INTEGRAL=y
WHEN THETA =xx.
FINISH.
```

Figure 5C

of integrals will be treated elsewhere. However, it is the purpose of the present note to point out that the techniques of the basic programming system and its recent extensions permit the input of mathematical text in a relatively free (and sloppy!) form. This text, in its internal representation, may be compiled for computation or manipulated symbolically to produce a neat form suitable for output and photo-reproduction. The essential point here is that these techniques permit adequate "typesetting" of mathematical text by relatively unskilled typists. Figure 6 represents *input* of five indefinite integrals. These have been typed so that numerators are not centered over denominators, portions are offcenter, and excessive or inappropriate spacing has been used. Figure 7 represents the output of a program that has automatically manipulated these expressions into a format suitable for photo-offset reproduction. As is illustrated in Figure 7(E), the program will automatically break up those lines of text, which are too long for one line, at the mathematically appropriate points.

In the basic programming system and in each of the above extensions, the user may type in any sequence. That is, he may arbitrarily space, backspace, superscript (move up), subscript (move down), overtype or output a code which indicates that an erasure is to be made. Thus he inputs an arbitrarily ordered string of primitive characters. This string must be reconstructed internally to correspond in some fashion to the typewritten document. The arbitrarily-sized (and possibly asymmetric) symbols must be recognized. Although the scanning of computer-stored text can be formalized as transformation rules on linear character-strings, we have found it convenient to consider the text's core image as a literal two-dimensional graph. The basic techniques for scanning along connected paths of the graph and through its nodes and along branches are simple and have been treated elsewhere.[11,15,17-20] Thus manipulations such as the elimination of unnecessary blank space can be ac-

(A) $$\int \frac{x^4}{(a^2+x^2)^2(\beta+x)}\,dx = \frac{\frac{a^4}{2(\beta^2+a^2)} + \frac{a^2\beta x}{2(\beta^2+a^2)}}{a^2+x^2} - \frac{a\beta(3\beta^2+a^2)}{2(\beta^2+a^2)^2}\,\text{TAN}^{-1}\frac{x}{a} + \frac{a^2(2\beta^2+a^2)}{2(\beta^2+a^2)^2}\,\text{LN}\left(a^2+x^2\right) + \frac{\beta^4}{2(\beta^2+a^2)^2}\,\text{LN}\left(\beta+x\right)^2$$

(B) IF $a^2 \neq \beta^2$ THEN $$\int \frac{x}{(a^2-x^2)(\beta+x)}\,dx = \frac{1}{2(\beta^2-a^2)}\left(\beta\,\text{LN}\left(\frac{(\beta+x)^2}{a^2-x^2}\right) - a\,\text{LN}\left|\frac{a+x}{a-x}\right|\right).$$

(C) $$\int (A+Bx)^3\,e^{kx}\,dx = \frac{1}{k}\,e^{kx}\left(\left(A+Bx\right)^3 - \frac{3B(A+Bx)^2}{k} + \frac{6B^2(A+Bx)}{k^2} - \frac{6B^3}{k^3}\right).$$

(D) $$\int \frac{1}{x^5}\,\text{TANH}^{-1}\frac{x}{A}\,dx = \frac{1}{4}\left(\frac{1}{A^4} - \frac{1}{x^4}\right)\text{TANH}^{-1}\frac{x}{A} - \frac{1}{12Ax^3} - \frac{1}{4A^3x}$$

(E) $$\int \frac{1\,dx}{x^5\,(A^2-x^2)^{3/2}} = -\left(\frac{1}{4A^2x^4(A^2-x^2)^{1/2}} + \frac{5}{8A^4x^2(A^2-x^2)^{1/2}} - \frac{15}{8A^6(A^2-x^2)^{1/2}} + \frac{15}{8A^7}\,\text{LN}\left|\frac{A+(A^2-x^2)^{1/2}}{x}\right|\right).$$

Figure 6

$$(A)\quad \int \frac{x^4}{(\alpha^2+x^2)^2(\beta+x)}\,dx = \frac{\frac{\alpha^4}{2(\beta^2+\alpha^2)} + \frac{\alpha^2\beta x}{2(\beta^2+\alpha^2)}}{\alpha^2+x^2} - \frac{\alpha\beta(3\beta^2+\alpha^2)}{2(\beta^2+\alpha^2)^2}\,\text{TAN}^{-1}\frac{x}{\alpha} + \frac{\alpha^2(2\beta^2+\alpha^2)}{2(\beta^2+\alpha^2)^2}\,\text{LN}\,(\alpha^2+x^2) + \frac{\beta^4}{2(\beta^2+\alpha^2)^2}\,\text{LN}\,(\beta+x)^2$$

$$(B)\quad \int \frac{x}{(\alpha^2-x^2)\,(\beta+x)}\,dx = \frac{1}{2(\beta^2-\alpha^2)}\left(\beta\,\text{LN}\left(\frac{(\beta+x)^2}{\alpha^2-x^2}\right) - \alpha\,\text{LN}\left|\frac{\alpha+x}{\alpha-x}\right|\right)$$

IF $\alpha^2 \neq \beta^2$

$$(C)\quad \int (A+Bx)^3 e^{kx}\,dx = \frac{1}{k} e^{kx}\left((A+Bx)^3 - \frac{3B\,(A+Bx)^2}{k} + 6B^2\,\frac{(A+Bx)}{k^2} - \frac{6B^3}{k^3}\right)$$

$$(D)\quad \int \frac{1}{x^5}\,\text{TANH}^{-1}\frac{x}{A}\,dx = \frac{1}{4}\left(\frac{1}{A^4} - \frac{1}{x^4}\right)\text{TANH}^{-1}\frac{x}{A} - \frac{1}{12Ax^3} - \frac{1}{4A^3x}$$

$$(E)\quad \int \frac{dx}{x^5(A^2-x^2)^{3/2}} = -\left(\frac{1}{4A^2x^4(A^2-x^2)^{1/2}} + \frac{5}{8A^4x^2(A^2-x^2)^{1/2}} - \frac{15}{8A^6(A^2-x^2)^{1/2}} + \frac{15}{8A^7}\,\text{LN}\left|\frac{A+(A^2-x^2)^{1/2}}{x}\right|\right)$$

Figure 7

complished by straightforward testing and commands which move only one character at a time. Since one is always working with a literal picture, at least as far as the system programmer is concerned, there is never any question of hidden coding ambiguities. The only question is whether the programmer has been clever enough to anticipate enough practical situations so that the system as a whole can be considered viable. Systems, such as these, are based on an *ad hoc* assemblage of straightforward strategies and are highly dependent for their success on the insight of the implementer and a correct assessment of the psychological traits of the user. Our experience has been that such systems can be brought rapidly to a state of working practicality in a relatively short period of field testing among uncommitted users.

Usage in programmed instruction

The field of Computer Assisted Instruction seems to be on the verge of a rapid expansion. One approach has been to use typewriter terminals with or without film projectors or CRT displays. Regardless of one's position on the feasibility or advisability of this kind of mechanical instruction, it would seem that capability for adding another degree of freedom for textual output can only improve the result. Since this can be accomplished at minor cost, it would seem a valuable extension to any such system. This would permit the construction of symbols and certain types of restricted "drawings" without increase in hardware complexity and with the use of programming techniques that have already demonstrated their feasibility.

Figure 8 is a fragment of a programmed lesson in differential and integral calculus which was written in the basic programming language. Essentially the student is offered a choice of three answers. He types his choice after the system types the problem and the word INPUT. If his answer is correct the system goes on to a problem of a different class. If his answer is wrong, it informs him which general formula is applicable to the problem and then outputs a similar example but with different parameters. It will keep varying the parameters on subsequent examples until a correct choice is made.

CONCLUSION

The point we wished to demonstrate in these various implementations is that it was possible to develop these perhaps elaborate methods using a relatively small com-

TYPE THE NUMBER OF THE CORRECT ANSWER AFTER THE WORD INPUT IS TYPED

$\frac{d}{dX} x^9 =$ (1) $9x^8$ OR (2) $8x^9$ OR (3) $8x^8$

INPUT 1

YOUR ANSWER IS CORRECT

$\int x^4 dX =$ (1) $\frac{x^3}{3}$ OR (2) $\frac{x^5}{5}$ OR (3) $\frac{x^4}{5}$

INPUT 3

WRONG, TRY AGAIN BUT REMEMBER THAT THE GENERAL FORMULA IS $\int x^n \, dX = \frac{x^{n+1}}{n+1}$

$\int x^7 dX =$ (1) $\frac{x^6}{6}$ OR (2) $\frac{x^8}{8}$ OR (3) $\frac{x^7}{8}$

INPUT 2

YOUR ANSWER IS CORRECT

$\frac{d}{dX} x^6 e^X =$ (1) $x^5 e^{-X}(6+X)$ OR (2) $x^6 e^X(6+X)$ OR (3) $x^5 e^X(6+X)$

INPUT 3

YOUR ANSWER IS CORRECT

Figure 8A

$\int \text{SIN}^3 X \text{ COS } X \, dX =$ (1) $\frac{\text{SIN}^4 X}{4}$ OR (2) $\frac{\text{SIN}^2 X}{2}$ OR (3) $\frac{\text{SIN}^3 X}{3}$

INPUT 1

YOUR ANSWER IS CORRECT

$\frac{d}{dX} \text{COS}^7 X \text{ SIN}^7 X =$ (1) $\text{COS}^6 X \text{ SIN}^6 X (\text{SIN } X + \text{COS } X)$ OR (2) $\text{COS}^6 X \text{ SIN}^6 X (7 \text{ COS}^2 X - 7 \text{ SIN}^2 X)$ OR (3) $\text{COS}^8 X \text{ SIN}^8 X (7 \text{ SIN } X - 7 \text{ COS } X)$

INPUT 2

YOUR ANSWER IS CORRECT

$\frac{d}{dX} \frac{1}{(1+X)^6} =$ (1) $\frac{-6}{(1+X)^5}$ OR (2) $\frac{6}{(1+X)^7}$ OR (3) $\frac{-6}{(1+X)^7}$

INPUT 1

WRONG, TRY AGAIN BUT REMEMBER THAT THE GENERAL FORMULA IS $\frac{d}{dX} \frac{1}{(1+X)^n} = \frac{-n}{(1+X)^{n+1}}$

$\frac{d}{dX} \frac{1}{(1+X)^3} =$ (1) $\frac{-3}{(1+X)^2}$ OR (2) $\frac{3}{(1+X)^4}$ OR (3) $\frac{-3}{(1+X)^4}$

INPUT 3

YOUR ANSWER IS CORRECT

Figure 8B

Special Edit Program
Operational Version 2 - December 1966
(Shaded characters indicate that they are to be typed in red)

The purpose of the edit program is to provide a character by character editing facility at the Flexowriter. The original program document which is to be edited is inserted into the Flexowriter. Do not turn the platen manually!

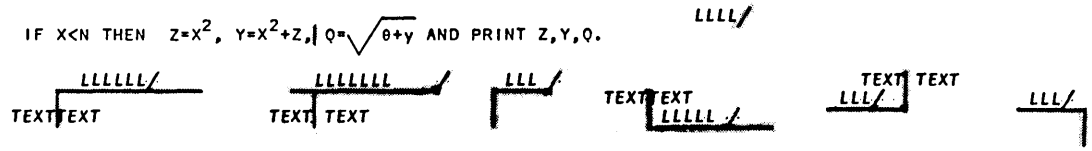
Initialize the edit operation by typing a red underscore under the first character in the program*.

To insert characters, phrases or statements, type them in the desired location.

To change any character(s) in a statement merely overtype with the desired character(s).

To erase any character in the statement, press the special erase button when positioned over the character or overtype with a red upper or lower case x. To erase a period, overtype with a red x. To delete an entire statement, type a red slash anywhere in the statement. (These may then be overtyped with the desired characters, phrases, or statements.)

Where there is not space to type directly into the program, an insert construction may be used. An insert construction consists of any figure constructed with an extensible red vertical, an (optional) extensible red horizontal, and a terminating red slash.

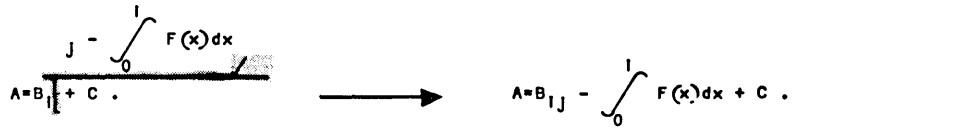


The first (in time) character typed in these constructions is a red vertical. The location of this red vertical determines where the left top-most character in the insert will be located in the program. The last (in time) character typed is a red slash which determines the right-most point of insert and terminates the insert. *LLLL* represents any typable characters.

Blanks may be inserted by using only those "empty" constructions where the slash is to the right of the vertical; the length of the blank will equal the empty space between the vertical and the slash.

Leading blanks to the first insert character are ignored but trailing blanks between the last insert character and the red slash are not.

Example:



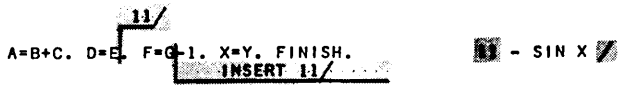
Insertion may be made or repeated by using the construction 

or 

where $0 \leq N \leq 9999$.

The insert may be typed anywhere on the document as N *LLLLLLLLL/* (*L* stands for any typable character). Either the insert or insert construction may be typed first and an insert may be referenced any number of times.

Example:



The program from this edit operation would then be: A=B+C. D=E - SIN X . F=G - SIN X -1. X=Y. FINISH.

To cancel the process of typing a new statement, type a red slash anywhere within the current typing. To erase an insert before typing a red slash, overtype the first red vertical with a red slash. [Once the insert is terminated, no further edit modifications may be made in the insert. Insert constructions may not be located above or below an underscore.] (The bracketed restrictions will be removed in the next program version.)

To delete an area from the program:  Note: Only the shaded area is deleted.

The area to be deleted is enclosed between red vertical lines and the deletion is terminated by a red slash typed in the two-dimensional statement between the verticals. (Only portions of one statement may be deleted at a time.)

The edit operation is terminated by typing a red F. If the edit operation is terminated and the program does not contain a "FINISH." statement, the edit program will insert "FINISH." after the last statement in the program.

In order to maintain proper vertical alignment, the user should realign the edit operation every few lines by overtyping in red any black letter, except F, X, x.

* When inserting the program page into the Flexowriter, possible horizontal skew may be tested by overtyping near the beginning and near the end of the first or subsequent lines before initializing.

Figure 9

puter (a GE-235). The extensions were particularly easy to implement by building on the assets of the basic programming system.

ACKNOWLEDGMENT

We extend our thanks to Jack May who participated in the development of the *IMAGE* statement and to Charles Amann for his assistance with the hardware aspects of this system.

REFERENCES

1 C AMANN M KLERER
Flexowriter/DIOA system
Columbia University Hudson Laboratories Technical Report no 124 Defense Doc Cntr Acq No 635 229 February 1966

2 C AMANN M KLERER
Documentation of a general purpose paper tape reader
Columbia University Hudson Laboratories Technical Report no 125 Defense Doc Cntr Acq no 635 379 March 1966

3 C AMANN M KLERER
Hardware documentation of a 127-button accessory keyboard
Columbia University Hudson Laboratories Technical Report no 126 Defense Doc Cntr Acq No (pending)

4 C AMANN M KLERER
Hardware documentation of an 8-button keyboard
Columbia University Hudson Laboratories Technical Report no 127 (in preparation).

5 K G BALKE G CARTER
The COLASL automatic system
Dig Tech Papers ACM Natl Conf pp 44-45 1962

6 A J T COLIN
Note on coding reverse polish expressions for single-address computers with one accumulator
Comput J vol 6 pp 67-68 1963

7 H J GAWLIK
MIRFAC: a compiler based on standard notation and plain English
Comm ACM vol 6 no 9 pp 545-547 1963

8 A A GRAU
The structure of an ALGOL translator
Oak Ridge Natl Lab Rept 3054 February 1961

9 M GREMS M O POST
A symbol coder for automatic documenting
Comput News vol 147 pp 9-18 and vol 148 pp 15-19 1959

10 C L HAMBLIN
Translation to and from polish notation
Comput J vol 5 pp 210-213 1962

11 M KLERER J MAY
Algorithms for analysis and translation of a special set of computable mathematical forms
Columbia University Hudson Laboratories Technical Report no 113 Defense Doc Cmtr Acq no 601 981 October 1963

12 M KLERER J MAY
An experiment in a user-oriented computer system
Comm ACM vol 7 no 5 pp 290-294 1964

13 M KLERER J MAY
A user oriented programming language
Comput J vol 2 no 2 pp 103-109 1965

14 M KLERER J MAY
Automatic dimensioning
Comm ACM vol 10 no 3 pp 165-166 1967

15 M KLERER J MAY
Recognition of arbitrarily sized and asymmetric typed symbols
Columbia University Hudson Laboratories Contribution no 232 July 1965 To be published

16 M KLERER J MAY
Two-dimensional programming
Proceedings of the FJCC 1965 vol 27 part I pp 63-75 Spartan Press Washington DC

17 M KLERER J MAY
Program documentation of a user-oriented programming system—part I
Columbia University Hudson Laboratories Technical Report no 118 Defense Doc Cntr Acq no 635 325 January 1966

18 M KLERER J MAY
Program documentation of a user-oriented programming system—part II
Columbia University Hudson Laboratories Technical Report no 119 Defense Doc Cntr Acq no 655 340 January 1966

19 M KLERER J MAY
Program documentation of a user-oriented programming system—part III
Columbia University Hudson Laboratories Contribu- Report no 120 Defense Doc Cntr Acq no 635 339 January 1966

20 M KLERER J MAY
Program documentation of a user-oriented programming system—part IV
Columbia University Hudson Laboratories Technical Report no 121 Defense Doc Cntr Acq no 635 341 January 1966

21 M KLERER
Hardware documentation of a user-oriented programming system
Columbia University Hudson Laboratories Technical Report no 123 Defense Doc Cntr Acq no 635 330 January 1966

22 J H KUNEY ET AL
Computerized typesetting of complex scientific material
Proceedings of the FJCC 1965 Spartan Press Washington DC

23 Los Alamos Scientific Laboratory
MANIAC II
Comm ACM vol 1 no 7 p 26 1958

24 W A MARTIN
Syntax and display of mathematical expressions
Project MAC Memorandum M-257 MIT 29 July 1965

25 M B WELLS
MADCAP: a scientific compiler for a displayed formula textbook language
Comm ACM vol 4 pp 31-36 1961

26 M B WELLS
Recent improvements in MADCAP
Comm ACM vol 6 pp 674-678 1963

27 A VANDERBURGH
The Lincoln keyboard—a typewriter keyboard designed for computers input flexibility
Comm ACM vol 1 no 7 p 4 1958

A graphic tablet display console for use under time-sharing

by L. GALLENSON
System Development Corporation
Santa Monica, California

INTRODUCTION

The RAND Tablet, a graphical man-machine communication device, is potentially one of the more fruitful approaches for two-dimensional graphic input to a computer. The high resolution of the tablet, high data transfer rates, and ease or "naturalness" of use are its chief assets. These same characteristics, however, give rise to the major problems in designing the tablet/computer interface. These problems are amplified when the interface is with a multi-accessed, time-shared computer.

A graphic tablet display (GTD) console has recently been designed to operate with the Q-32 Time-Sharing System (TSS) at System Development Corporation. The successful completion of the task has provided new insights into system design problems involving highly interactive consoles in a time-sharing environment; some interesting innovations in the design of a graphic tablet console; and some solutions to graphic tablet interface problems.

The TSS at SDC has been well documented.[1] Understanding the operation of the graphic tablet display console does not necessarily require a complete understanding of TSS. Suffice it to say that TSS affords the capabilities of a large digital computer simultaneously to a number of users, each of whom assumes he has the full capabilities of the computer system.

For the typical TSS user's console—a keyboard/printer—to be considered interactive, it must receive some response from TSS in fractions of a second, and a certain amount of computational response within several seconds. With the graphic tablet display console, however, TSS reaction time must be on the order of a few milliseconds, and computational response on the order of one second. The high data transfer rates and the speed of response required for user psychological reasons demand that the user-to-system interface be "tightly coupled."

The trend in modern computer system design is to provide external I/O buffers for CRT and graphic consoles; thus data is not readily accessible to the object (user) program. Typical systems require block transfers of all the data from the console into main core for processing, maintenance of the complete image of display tables within the user's core space (or complicated algorithms for regenerating the tables), and finally, block transfers back to the I/O device. In designing an interface for a highly interactive console to these computer systems, the I/O transfer time can become an appreciable portion of the required interaction time. The system configuration described in this paper provides an I/O buffer area in core memory that is directly accessible to both the central processor and the console, thus placing the data in a display buffer very "close" to the object program.

TSS/GTD general system description

The major components of the Time-Sharing System (TSS) are the AN/FSQ-32 computer, a PDP-1 computer, and the time-sharing executive. The Q-32 is the main processor; the PDP-1 serves as an I/O processor for all the interactive I/O devices. The two computers are coupled via a core memory, called Input Memory (IM), which is part of the Q-32 and is directly addressable by both computers. IM is a core bank identical to the other Q-32 main memory modules (16,384 48-bit words with a 2.5-μsec cycle time). It is accessed by the Q-32 for reading or writing *data,* but *programs* cannot be executed from it. TSS protects IM with write-protect logic from the Q-32; therefore only the executive can write into IM.

To the PDP-1, IM appears as 32,768 18-bit words of main memory extension (since the PDP-1 has 18-bit words). Thus, 6 bits per half word of IM are potentially lost to the PDP-1. To avoid the loss of memory capacity, two special instructions—"load byte" and "deposit byte"—have been added to the

PDP-1. These instructions allow the PDP-1 to access *all* of IM with 6-bit bytes for data fetch and store cycles. Using an I/O preprocessor to store data in a core buffer that is directly addressable by the main processor has proven to be a powerful device for implementing TSS, and has facilitated the interfacing of the graphic tablet display console.

The PDP-1 initiates service to all interactive consoles in response to an interrupt from the console. The processing of the interrupt causes the execution of routines unique to the class of console requesting service. In general, the routines transfer the data in or out, buffer the data, translate to appropriate codes, examine the data for end-of-message cues, and inform the Q-32 of user requests.

The use of a separate processor for all interactive data I/O for TSS has given SDC the ability to interface a large variety of devices with TSS. These include teletypes, special keyboards and typewriters, lightpens for displays, special pushbuttons, high-speed data lines for computer-to-computer communications, automatic dial-up units, as well as the graphic tablet display console. To provide quick response time to the user of the GTD console, the PDP-1 performs a significant amount of preprocessing on the raw data. A project is currently underway to study the tradeoffs of allowing the slower PDP-1 or the faster Q-32 to perform the required user functions. Although the user cannot directly modify the PDP-1 executive interactively, a system programmer can modify it to meet new user requirements.

Operation of GTD console

The operation of the GTD console in TSS can best be described in the context of a user of the console. One such user at SDC is involved in a research program that is attempting to provide on-line computer recognition of characters written on the RAND Tablet.[2] The user can draw, or print, characters and observe the "inking" on the tablet surface (see Figure 1). After removing the pen from the tablet, TSS responds by displaying the symbol(s) as interpreted by the object program. This process is performed by a hardware/software interface between the GTD console and the TSS (PSP-1). Figure 2 depicts the major data paths to and from the GTD console.

Hardware operation

Inputs to the PDP-1 are initiated by interrupts from the console to the sequence break system. The interrupt causes the PDP-1 to execute routines for preprocessing the data from the console. A variable strobe clock (which determines the interrupt rate of the tablet) is provided to minimize the buffering and processing requirements. A 2- or 4-millisecond rate (500 or 250 points-per-second) is adequate for present users at SDC. A second oscillator (30 Hz), effective only when the pen is up, also initiates interrupts to the PDP-1 so that pen coordinate data for positioning information and pen switch time-out can be processed. All interrupts from the console are ignored by the software if the display is not being used or if an object program using the console is not loaded in TSS. Input data are preprocessed by the PDP-1 and stored in the display core buffer located in Input Memory

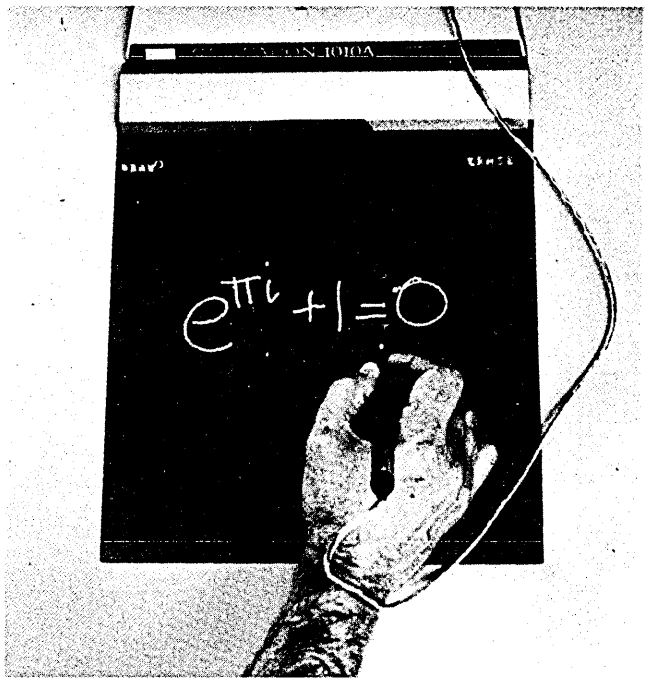


Figure 1—Graphic tablet display in operation

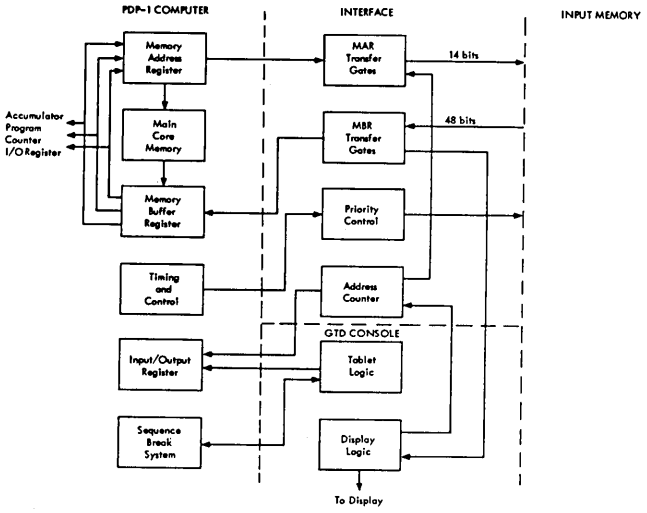


Figure 2—GTD console interface block diagram

(IM). The display buffer is continuously read by the display control logic; thus immediate visual feedback is given to the user.

The display portion of the console interface consists of memory access logic to IM, data transfer lines, and priority logic to handle PDP-1 and display memory access conflicts. The memory access logic is essentially a counter, modulo 2048, stepped for each display word cycle. The counter is used to step through the core address within the block of IM dedicated to the GTD buffer. The average display word cycle is 15 microseconds or 30 milliseconds per display frame.

The display uses 24 bits of the 48-bit word in core (two display words per IM word) for each display cycle, so that gating is provided to transfer the proper half word to the console. The address counter is also available to the I/O register of the PDP-1, so that when using the stylus in the "light-pen mode," the address of the displayed data is made available to the user.

The display buffer is 1024 words (48 bits) of core in IM and contains the processed display table, the stylus input data, as well as several control words and indicators. For example, the pen input contains coordinate data which are displayed in the left 24 bits; the point-count, start-of-segment indicator and other control bits are in the right half word. The user has control of almost all the 2,048 display words available within the table, and determines how the buffers are divided between the stylus inputs and display.

The IM has three potential users: the Q-32 central processor, the PDP-1, and the display, so that logic is required to control the traffic problem. A priority scheme of CPU, PDP-1, and console (highest to lowest) has been established, and no device is capable of usurping all the core cycle time. Actually, programming conventions for the PDP-1 and the Q-32 are such that the problem of one device requiring continuous service from IM is never encountered. The priority logic is required, however, to take care of simultaneous requests for core memory.

Software operation

Processing for the GTD is initiated by an interrupt to the PDP-1 for pen tracking or pen data inputs (see Figure 3). Pen tracking occurs at a rate of 30 Hz while the pen switch is open. The x and y positions of the pen are read and placed into one register of the buffer table. Pen data inputs occur while the pen is in use (i.e., the switch is closed). The functions performed by the PDP-1 in preprocessing the raw data are as follows:

1. Filtering: a point must move at least 3 increments in x or y before it is called a new point; 500 points-per-second are examined.
2. Point count: a count of the number of times a point remains within the filter window is stored with the point coordinates.
3. New stroke: setting a control bit within the display word indicates that the point is the start of a new stroke.
4. Min/max: each stroke is described with an x and y minimum and maximum at the end of the stroke. (This feature is available in one version of the PDP-1 program, but is not shown in Figure 3.)
5. "Inking": new pen data are placed in the display buffer in an appropriate format for displaying.
6. Pen up/down: the tablet is read periodically to determine if the stylus is in use.
7. Smoothing: new pen data are averaged over the last eight inputs, prior to filtering.

In addition, a certain amount of bookkeeping of the display buffer is performed. The processing of pen data consumes 30 percent of PDP-1 time when reading 500 points-per-second. The PDP-1 time required for pen tracking (pen up) is insignificant. Further preprocessing capabilities will be introduced into the software as requirements dictate, for example, the capability of selectively erasing a single point or character on command from the user.

When input from the GTD is complete, the PDP-1 alerts the Q-32 executive to request processing. Completion is noted by either of two events: (1) when the tablet buffer is full (no "ink"), or (2) when the pen has not been used for a predetermined period of time after an input. The Q-32 executive then schedules the user's object program. The processing of the data is performed by the object program directly accessing the data in IM.

At the completion of the processing by the object program, the display table is updated. New display information is added, deletions are made, and the control words are reset. The Q-32 executive must transfer the table from main core to IM, since the IM is write-protected from all users (a number of sensitive tables and data for TSS also reside there). The display table contains control words used by the PDP-1 program. These include a pointer to a starting address for the input of new pen coordinate data, and an octal number (from 1 to 37) representing the delay time in ¼ seconds that the pen is allowed to be inactive prior to an input-complete signal.

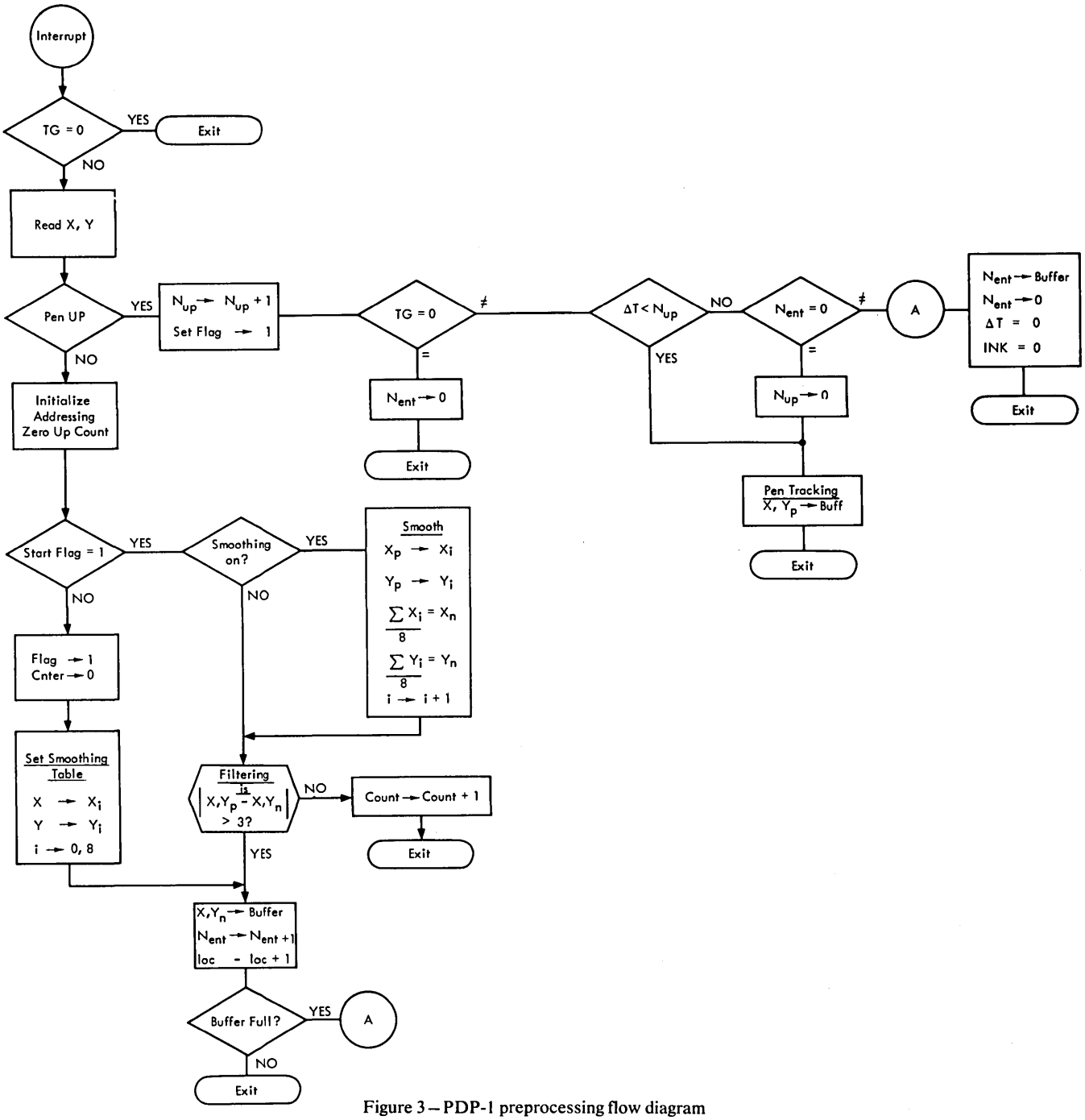


Figure 3 – PDP-1 preprocessing flow diagram

GTD console components

The graphic tablet display console (see Figure 4) consists of a RAND Tablet (Grafacon Model 1010A built by Bolt, Baranek and Newman),* an input stylus, and an oscilloscope and lens system for rear-projecting the CRT image on the tablet surface. The rear-projection technique was originally suggested by the inventors of the RAND Tablet – Davis and Ellis[3] – and was developed at SDC. It produces the effect of a "live piece of paper," which was the intent of the tablet inventors. They maintained that a direct-viewing oscilloscope adjacent to the tablet was desirable and that users normally adjusted to the problem of conceptually superimposing the pen and "ink" in a matter of minutes. Experience with the tablet at SDC indicates, however, that a significant number of people have difficulty using the tablet in that manner.

*Graphic tablets are also currently available from other vendors.

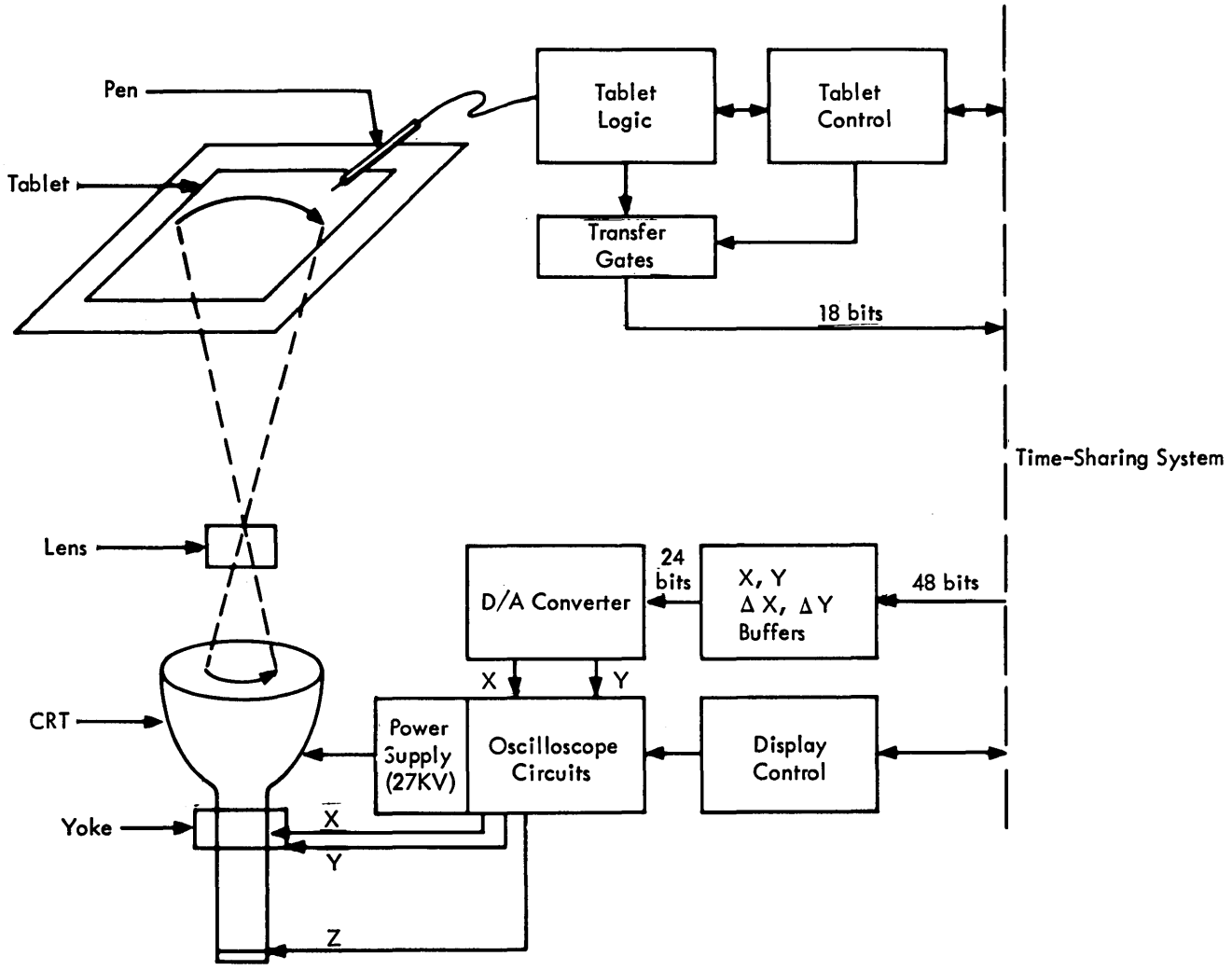


Figure 4 – GTD console components

Tablet

The superposition of the display on the tablet surface is a natural evolution and makes the displayed feedback more meaningful to the user, as well as easier to use. The tablet is operated by pressing the stylus pen on the tablet surface, thus closing a microswitch contained within the pen. The pressure necessary to activate the switch is equivalent to that normally required to write with a ball-point pen. The tablet electronics reads some 4,500 points-per-second, which proves to be more than adequate for all current applications. The amount of input data can be reduced – thus reducing the size of the input buffer required – by introducing a variable oscillator used for generating the PDP-1 interrupts.

Typically, the tablet is read by the PDP-1 every 2 or 4 milliseconds, 250 to 500 points-per-second. The tablet electronics provides a 22-bit buffer register (10 x-bits, 10 y-bits, and 2 control-bits) to hold the stylus position information until read by the computer. The PDP-1 is an 18-bit machine, so that at least two input read cycles are required. The extra bits gained in this 36-bit transfer are used to provide additional information to the user. These include data for:

- The setting of the read interrupt clock.
- Identifying the first point of a new stroke (pen down after a previous pen up).
- Pen switch down.
- Foot switch down (not used).

All this information can be made available to the user by the PDP-1 preprocessing. However, the PDP-1 typically ignores some of this data at the discretion of the user.

Display

The display function is performed by an oscilloscope with appropriate buffer, digital-to-analog converter, and control logic for handling the digital data and driving a projection-type CRT. There are no special requirements for the oscilloscope; the tolerable drift, linearity, and amplification for driving the CRT are well within the state of the art. The logic for producing the analog voltages, buffering and unblanking signals is also typical of currently designed displays using random positioning combined with a delta positioning mode for special features.

The display uses a 24-bit word, 10 bits each for the x and y positions, an unblanking bit, and 3 bits for mode control. The Δx and Δy data are represented by 3 bits each and are primarily used for generating characters within a 5-by-7 dot matrix. The data is double-buffered so that movement of the CRT beam between successive points is minimized, reducing the settling time of the beam and increasing the speed of operation. The second buffer is also used to store the positioning data when in the character mode to provide the reference for subsequent delta positioning information. The average display cycle time per 24-bit word is 15 μsec, which includes 4 μsec for unblanking the CRT. Consideration is currently being given to improving the display capability with vector, circle and arc-generating logic.

The CRT used for projection is a 5ZP4, and was selected for convenience. No appreciable amount of effort has yet been expended in surveying the market for a better performing CRT for this application, since the current one seems adequate. The 5ZP4 operating at 27,000 volts anode potential is predicted to have a 5-mil spot size and can produce a spot of 300 foot lamberts when operating with 30 microamps anode current unblanked for 4 μsec. The CRT raster is limited to 2.2 inches to minimize effects of the CRT distortion (pincushioning and barrelling).

The CRT image is projected through appropriate optics onto a rear-projection screen which is built into the front surface, or wearcoat, of the tablet. The projected image is 10 by 10 inches—the tablet size—which calls for a 4.5 magnification in the optics. The net effect produces an image of 15 foot lamberts and a spot size of 20 mils. The light loss through the printed circuit wires/screen combination is 75 percent, and the remaining loss of brightness is due to the optics. The display's visual characteristics are produced with an average 50-Hz refresh rate; the results are comparable to those obtained with the direct viewing displays SDC has been using with the graphic tablet.

A recent innovation (which is being incorporated in the GTD console at SDC) was developed at The RAND Corporation. "Comparator" logic was designed that can compare (within preset limits) the analog x and y voltages of the display with the digital coordinates of the tablet. This provides the user a "light-pen" capability with the stylus and tablet. The user can point close to a spot being displayed and receive positive indication about the point of interest from the display table rather than from inputs from the pen. In the GTD console, the comparator will be used to generate an interrupt pulse to the PDP-1. The PDP-1 can then read the address counter of the display, providing the object program the address of the word being displayed rather that the x and y positions.

CONCLUSION

A graphic tablet display (GTD) console embedded in a time-sharing system can provide a highly effective means for direct input of graphic information to a computer. Such a device, employing a RAND Tablet with a rear-projection display, has recently been developed at SDC. The interface makes the input and display buffer directly accessible to the object program, which enhances the response time and is efficient in use of core memory. The interface takes advantage of the previously designed interface of the PDP-1 and Q-32 computers. Thus, integrating the GTD console into the system was relatively simple.

Experience at SDC—where a variety of I/O devices have been installed with TSS—has demonstrated the advantages of an I/O preprocessor for interfacing interactive consoles. Although there are limitations as to the amount of preprocessing the PDP-1 can perform (due to space and time restrictions, compounded by the limitations of a small machine), the I/O preprocessor does provide another important degree of freedom for the system designer.

Employing the tablet display as a projection on the writing surface has improved the "naturalness" of this I/O device and has encouraged more users. The display hardware is more expensive for equivalent quality of direct-viewing displays, but the additional advantages gained may be justified for given applications. Additional innovations can be introduced to improve the flexibility of the GTD configuration. These include: the projection of slides in conjunction with the CRT image for tablet manipulation; the adding of a color wheel to achieve computer-generated

color displays; and the use of a camera to achieve hard copy output from the CRT. Significant improvement in light output of the projected CRT image must be achieved before these additional capabilities are introduced.

The GTD console described in this paper has been successfully demonstrated. As yet, it has not achieved its full capabilities as a tool for man-machine communication. Currently, several interesting programs are being developed at SDC using the GTD console, including the one described above—computer recognition of hand-printed characters. The intent of the GTD design and interface is to provide sufficient flexibility and ease of use to encourage a variety of users to employ this new, powerful tool. Efforts are currently underway to design a similar console and interface for an off-the-shelf, third-generation computer.

ACKNOWLEDGMENTS

The author wishes to express his gratitude to Mr. Clay Fox and Mr. Morton Berstein of SDC for their contributions to this project. Mr. Fox provided PDP-1 programming support; Mr. Berstein is an imaginative user of the console, who suggested many of the system features.

REFERENCES

1 J I SCHWARTZ E G COFFMAN C WEISSMAN
A general-purpose time-sharing system
Proceedings Spring Joint Computer Conference pp 397-411 1964

2 M I BERNSTEIN
An on-line system for utilizing hand-printed input
SDC document TM-3052 19 pp July 1966

3 M R DAVIS T O ELLIS
The RAND tablet: A man-machine graphical communication device
Proceedings Fall Joint Computer Conference pp 325-350 1964

Multi–function graphics for a large computer system

by CARL CHRISTENSEN and ELLIOT N. PINSON
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

I. INTRODUCTION

Although much effort has been spent trying to improve communications across the man-computer boundary, most large computer centers still rely on punched-card-input and line-printer-output for the bulk of their load. Even modern "time-shared" systems rely principally on teletypewriter alphanumeric communication between man and machine.

A great deal has been learned, however, about other ways of communicating with computers, particularly in terms of graphical information.[1,2,3] On output, graphs and drawings convey meaning to a human viewer much faster than large tables of numbers. Computer generated motion pictures add a time dimension to a display that can help provide physical insight into complex problems.[4,5] On input, the ability to identify objects in a picture by pointing at them, or to modify a picture in a natural way by drawing in it is a great convenience.[6,7]

Graphical output has been an important way of presenting results to Bell Laboratories' computer users since the fall of 1961, when a Stromberg Carlson 4020 microfilm printer was installed at Murray Hill. It has allowed users to replace pages of tabular data with pictures that are far easier to understand. Microfilm picture generation was running recently at a rate of about 500,000 frames per year. In addition, computer movies were being generated at a rate of about 1,000,000 frames per year.

In 1964 the experimental GRAPHIC-1 display terminal[8] was interfaced to a 7094 at Murray Hill to provide semi-on-line graphical input-output to the 7094 for a single user at a time. Despite the fact that this was a one-of-a-kind system, it was extremely popular and heavily used. It incorporated a small processor in the terminal to handle all real time requirements generated by the user.

More recently, an experimental on-line graphical output device that can produce hard-copy at a rate of about one page every five seconds was interfaced to the 7094. This device uses a microfilm intermediate stage with a rapid high temperature developing process to produce output within 30 seconds of the film being exposed. This, too, has been popular with users, although mechanical problems have caused excessive down-time for the device.

Although the graphics facilities on the 7094 system were extremely useful, they were far from what a modern, large-scale computing center should offer its customers. Among the features that seemed ready for improvement were the hard-copy output turnaround time (frequently overnight on the SC-4020, although sometimes as good as a few hours), the fixed geographical position of all devices, the inaccessibility of the system through standard transmission facilities, and the extremely small number (in fact, just one) of terminals from which users could run their graphical programs.

In Section II of this paper an overview is given of a new graphics system being developed at Bell Laboratories. Following sections describe certain aspects of the system in more detail. Section III describes hardware, and Section IV describes the Graphical Data Structure used to represent picture information in the computer. The GRIN-2 graphical programming language is discussed in Section V (an annotated example of a GRIN-2 program is given in the Appendix). Section VI describes some functions performed by the GRAPHIC-2 Executive System, and Section VII describes briefly some of the functions of the GE-645 Software System. A brief summary is given in Section VIII.

II. System overview

A modern large scale computer facility should provide a variety of graphical services for its customers. These range from hard-copy output of high quality to highly interactive on-line graphical communication be-

tween man and computer. The graphical services provided by the system described in this paper include:

1. Rapid hard-copy output (STARE system);
2. On-line direct-view cathode ray tube output (GLANCE system) as an accessory display for a teletype console;
3. Remote on-line interactive graphical input-output (GRAPHIC-2 system);
4. Precision microfilm and hard-copy output;
5. High precision drafting quality output.

Obviously, no single type of display can provide these very different services in an economical way. Rather, several different kinds of equipment are required. Services 4 and 5, although very important, will not be discussed further since they are provided by standard commercially available equipment, namely, an SC-4060 microfilm printer and a Gerber drafting table. As will be seen below, however, programming support for these devices is handled in large part by software used in common for all hardware systems. The first three categories involve systems for which a combined hardware-software design effort was performed. Hardware details for each of these systems (STARE, GLANCE, and GRAPHIC-2) are given in Section III.

The graphics systems will run on a large multi-access computer installation at Bell Laboratories' Murray Hill facility. This computer system will soon be operating under the Multics[9-13] system. The equipment features a dual-processor GE-645 computer with 256K of 36-bit 1 μs core memory and extensive on-line mass storage (drum, disk, etc.). Multics system features include a large on-line multi-level file system, memory segmentation and paging (to give each user an extremely large virtual memory space), and the ability to communicate with many on-line communication devices. The system will simultaneously service many console users and programs entered in a conventional "over-the-counter" mode. Most on-line consoles will be of the teletypewriter variety. Additional graphic consoles as described in this paper will also be available. Response times for users of the graphics system will be measured in seconds rather than minutes (for GRAPHIC-1) or hours (for the SC-4020). True remote access to the computer via dialed up voice grade phone lines will be possible from several of the terminals.

For computer graphics to have a substantial impact on many people, it is necessary to make graphic equipment available and accessible. Two factors must be considered: first, physical accessibility, made possible by providing multiple devices which can be used simultaneously and which are conveniently located; second, program accessibility made possible by providing the user with a powerful set of system functions and a higher level language for his graphical programming (see Section V on the GRIN-2 language). Although specific numbers of devices are not known at this time, Figure 1 shows a configuration that should be achieved during 1968.

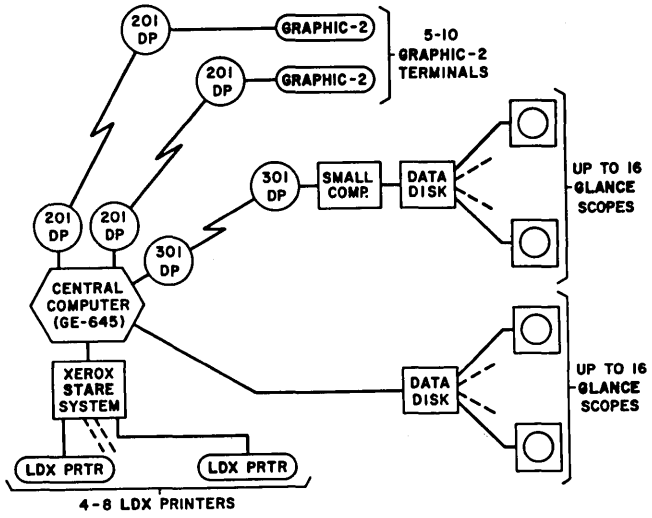


Figure 1—On-line computer graphics facility

Despite the fact that many different types of hardware are supported, the bulk of the software in the central computer is device independent. This is because a device independent representation of picture information is used in the central machine. This representation, called the Graphical Data Structure (see Section IV) describes a picture that can be displayed on any of the graphical terminals in the system. The Graphic Data Structure provides a standard for communcation of picture information between programs and is the form in which pictures are stored in the GE-645 File System. Device dependent Translator modules are used to convert from the central format to the command formats required by a particular device. There is one such Translator module for each type of graphic device in the system.

In the case of the STARE hard-copy system, a user will be able to go to an output station near his office rather than have to go to a single central distribution point. Up to eight such remote printer stations may be served by the hard-copy system. Among other uses, the rapid hard-copy facilities are expected to aid users accessing the machine from teletype consoles whose low speed makes them unsuitable for listing alphanumeric files. Dumps and other listings can be produced on the STARE system upon command from a teletypewriter console.

The intent of the GLANCE on-line direct view CRT system is to provide an accessory graphical display capability for standard teletype consoles. The CRT dis-

play is physically independent of the teletypewriter, although located immediately adjacent to it. Under software control, graphical output from the program being controlled from the teletype will be displayed on the CRT display. The CRT display can be maintained without consuming costly central machine resources, since a special high-speed disk buffer is used to store the commands that generate the picture. A track of this disk has to be loaded once for each picture to be shown. Thereafter, the picture is maintained on the display tube without central machine intervention. Since a high data rate is used between the disk buffer and the displays it supports, each display must be within coaxial cable reach of its buffer. A complete GLANCE system (disk plus CRT terminals) can be located remote from the central computer if a small computer is added to the system. The small computer can receive coded graphical data at low speed over a transmission link, convert it to the command set required by the display logic, and write a picture at a time at high speed onto the magnetic disk. An example of this type of operation is shown in Figure 1.

The GRAPHIC-2 system is an interactive graphic input-output terminal. Graphical input is accomplished by using a light pen to point at objects displayed on a CRT. This system accesses the central computer over a voice grade transmission line. One of the principal objectives of the GRAPHIC-2 system is to provide good terminal response to the operator while using slow (but economical) communication lines to the central computer system. Unfortunately good terminal response is measured in fractions of a second, while response from the central computer may take several seconds. For example, transmission of a 100-word data block (18 bit words) takes almost one second over the 2000-baud voice grade line. Added to this must be the delay involved in capturing the central processor (which is simultaneously trying to service numerous other users) and in carrying out central machine computations.

The solution to this problem lies in having GRAPHIC-2 itself handle users' real time demands. Thus, such actions as pen-tracking and drawing are handled in real time at the display terminal (by the PDP-9 processor that is part of the GRAPHIC-2). It is only the results of these actions (final X-Y coordinates or identity of a selected object) that are sent back to the central computer, where a master copy of the picture is updated. Fortunately, modifications to this master copy do not have to be made in real time since the user at the terminal sees the results of his actions on the local version immediately.

Two versions of a user's program are compiled. These programs originate from a single GRIN-2 (see Section V) source program. Both compilations are performed by the GE-645. One produces GE-645 machine code and the other produces PDP-9 machine code. The former executes in the GE-645, operates on the Graphic Data Structure (see Section IV) in the central machine and obtains its "pseudo real-time" inputs from a queue sent back from GRAPHIC-2. The latter executes in a GRAPHIC-2 terminal, modifies a local (at GRAPHIC-2) copy of the data structure and services real-time inputs as they occur (light-pen strikes, button pushes, keyboard inputs), simultaneously recording their occurrence on the queue destined for the GE-645. The queue is periodically sent to the central computer, allowing the GE-645 execution of the user's program to catch up with GRAPHIC-2. This strategy minimizes the amount of data that have to be transmitted from GRAPHIC-2 to the central machine, since only the real time events are transmitted.

III. Display hardware systems

The following three display systems are described:

A. The STARE (Short Turn-Around REcorder) rapid hard-copy system.
B. The GLANCE (GraphicaL Accessory oN-line ConsolE) cathode ray tube output display.
C. GRAPHIC-2, a graphical input-output display terminal that includes a small processor, display CRT, light-pen and keyboard.

A. The STARE hard-copy system

The STARE hard-copy facility accommodates up to eight rapid hard-copy output stations. The organization of the system is shown in Figure 2. The system consists of multiple remote Xerox LDX (Long Distance Xerography) Printers interfaced to a central buffer memory and control unit that is located near the GE-645. The LDX printers form an image by a raster scan technique,

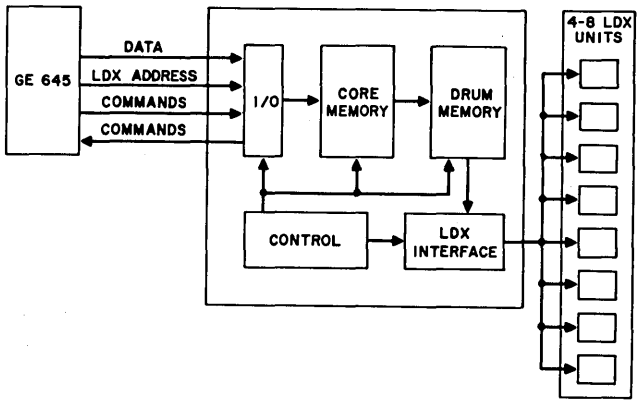


Figure 2—STARE system organization

so the data from the central buffer unit to the printers are essentially a video signal for controlling the intensity along each scan line. These data to the LDX Printer are transmitted from the central buffer at about a 250,000-baud date over coavial cable links. There are eight separate buffer areas on the magnetic drum which function independently. Therefore, transmission to up to eight printers can take place simultaneously.

The central buffer unit is a digital scan-converter. It accepts drawing commands in an incremental command code (the same one used by the GLANCE system described below) and converts these to the raster scan required by the LDX Printers. The complete picture is specified on a plotting grid 1024 × 1408 (corresponding to a plotting area of 8″ × 11″). The core memory contains enough storage to represent the raster scan of about a one inch strip of the output (128 lines by 1024 bits/line). An I/O command from the central computer specifies which strip the core buffer is to scan-convert. The incremental commands that specify the entire picture are then sent to the buffer unit. The buffer logic transforms these to raster scan bits for the proper strip and stores them in the core buffer, 1 bits representing black dots and 0 bits representing white. After the picture has been scan-converted, the core image is automatically transferred to the appropriate tracks on the magnetic drum. This process is repeated eleven times to transfer the entire picture to the drum. An I/O Print Command is then issued to the computer which starts the appropriate LDX Printer and initiates the drum-to-printer transfer.

The data rate between the central computer and the core buffer unit is about $2x10^6$ baud. Since a complete picture is often specified in under 10^5 bits in the incremental code, each burst of data takes about .05 seconds for the core to core transfer. The buffer-core to drum transfer is at about a 10^6 baud rate, and takes about .15 seconds to complete (including drum latency). A full picture (11 strips), by these rough calculations, takes about 2.2 seconds to transmit from central computer to drum buffer. About an additional five seconds are required for the drum to LDX Printer transfer. Thus, a single printer can produce about 8 frames per minute in this system, and a maximum throughput of over 24 frames per minute distributed among several printer stations is possible.

B. The GLANCE direct-view scope system

The GLANCE system has been described in a previous paper.[14] However, the essential elements will be repeated here for the reader's convenience. Figure 3 shows the GLANCE configuration. A central disk buffer holds picture information for up to 16 GLANCE display terminals. A single track of the disk is dedicated

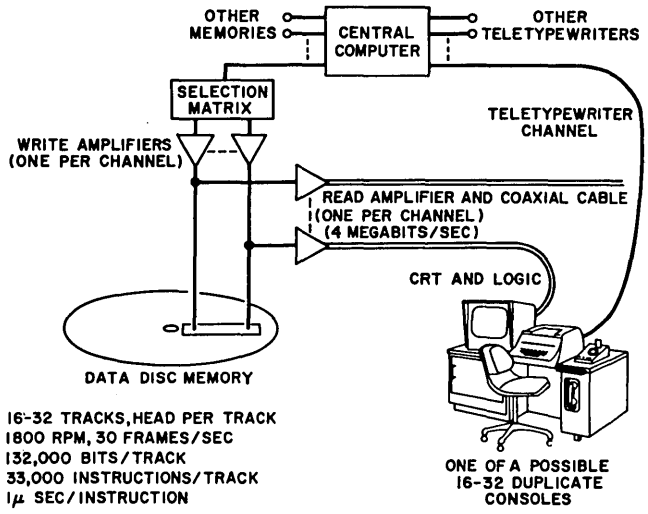
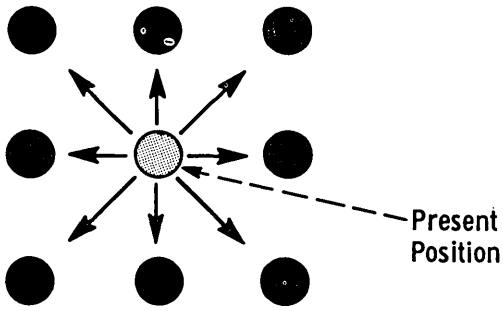


Figure 3—GLANCE system organization

to each of the terminals, and each of the tracks has its own read-head and line driver. The picture on each track is specified in terms of the simple four-bit incremental commands shown in Figure 4. These commands are decoded by logic in each of the scope terminals where X-Y deflection and unblanking signals are produced to position and display points. The commands allow the beam to be moved a certain number of units in any direction on a 1024 × 1024 grid (left-right, up-down, or diagonally) covering a 10″ × 10″ area on



COMMANDS

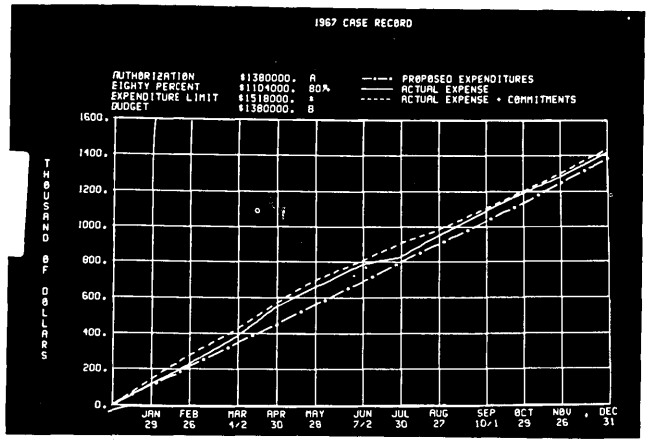
Control	Moves
No op	Right
Reverse beam on/off	Right-Up
Increment intensity (mod. 4)	Up
Set scale x 1	Up-Left
Set scale x 2	Left
Set scale x 4	Left-Down
Set scale x 8	Down
Set scale x16	Down-Right

Figure 4—Incremental command plotting technique

the scope-face. Moves can be made with the beam blanked to reposition it without leaving a trace. Four spot intensities can be used. A scale-factor can be preset so that incremental commands cause a beam move of one, two, four, eight, or sixteen grid units. This permits fast slewing with the beam blanked.

The disk rotates at 30 revolutions per second (1800 rpm), so the picture on each scope is refreshed at a 30-cycle rate. With the proper phosphor this rate is high enough to give a flicker-free display. The data transfer rate from disk to display scope is $4x10^6$ baud, so this connection is made with coaxial cable. The display terminals can be located up to a mile or so cable length from the central disk buffer. At the data rate used, 10^6 points/second are plotted by the display terminal. Figure 5 shows a picture of a GLANCE console including a Teletype, and Figure 6 shows pictures taken directly from a GLANCE CRT.

Figure 5—The GLANCE terminal



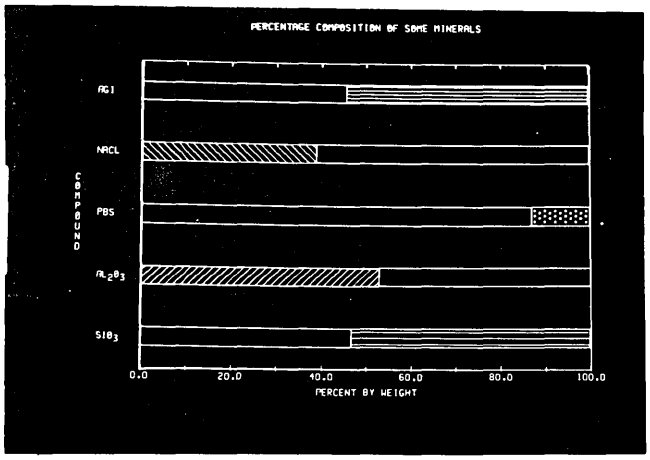


Figure 6—Sample pictures from a GLANCE CRT

C. GRAPHIC-2

Many of the factors that affected the design of GRAPHIC-2 were discussed in Section II of this paper. A detailed description of the system is presented in a forthcoming paper.[15] Only the essential elements will be repeated here. Figure 7 shows the GRAPHIC-2 organization. A picture of the hardware is shown in Figure 8. A standard DEC PDP-9 with optional automatic priority interrupt and a direct memory access channel is the heart of the system. Its memory consists of 8K words of 18-bit one microsecond core. This memory serves as display buffer memory and also holds programs that execute in the PDP-9. An optical paper tape reader and a paper tape punch are standard. The display scope and display command set are described in detail in the forthcoming paper.[15] A light-pen and keyboard (ASCII alphanumerics plus 8 pushbuttons) complete the GRAPHIC-2 hardware.

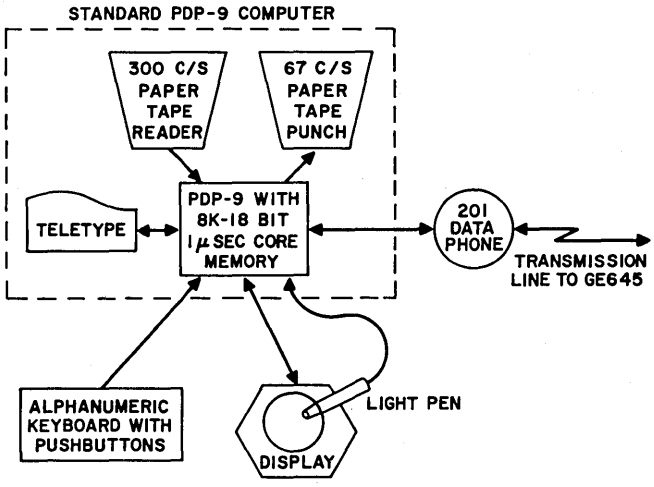


Figure 7—GRAPHIC-2 organization

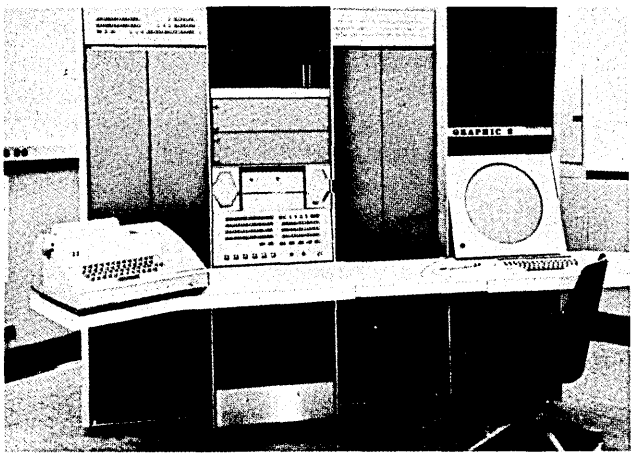


Figure 8—The GRAPHIC-2 terminal

The scope has a character generator that can display the full ASCII 96 graphic set. Among the features that have been included in the scope hardware to facilitate programming are: Edge violation traps whenever a command tries to make a line cross a scope boundary, either in or out; software commands to determine the exact state of the scope at any time and to restore the scope to any state; commands which when encountered automatically stop the scope and interrupt the PDP-9 processor. In short, a great deal of thought was devoted to the design of a scope-processor system that is easily programmable.

IV. The graphic data structure

The various graphical devices that are supported by GRAFCOM (GE 645 Graphical Communication software) function quite differently at the hardware level. In particular, they use different command formats to generate pictures. However, in the GE-645 a single standard way of describing graphical information is used that is independent of any particular display device. This common representation is called the Graphic Data Structure. All picture building routines generate blocks that are attached to the user's graphic data structure. All picture editing routines assume they are operating on pictures represented in this standard way. Thus a single set of graphical programs is used to manipulate graphical information whether a GRAPHIC-2 interactive console, or any other graphic output device is being used. Before display commands are sent to a particular type of device, a device dependent program is called to convert from the standard format to the special command set used by the device. Methods for representing picture structure have been described in the literature.[1,16,17] The structure used in GRAFCOM differs from these in detail, but was strongly influenced by several previous systems.

Although the main graphical data base is in the large central computer, the GRAPHIC-2 terminal stores picture information in an identically structured form. As will be seen, information in the graphic data structure is stored in discrete blocks that are linked together by chains of pointers. Although formats within blocks in the GE-645 are different from those in the GRAPHIC-2, there is a one-to-one correspondence between blocks in the two structures. Because the memory in the GRAPHIC-2 is small (8K, 18-bit words), all blocks in the data structure do not have to be present in GRAPHIC-2 at the same time. A dynamic memory management system fetches blocks from the central computer when they are needed, and releases memory in GRAPHIC-2 to free storage when blocks are no longer needed. Some differences between block formats in the central machine and in the GRAPHIC-2 are explained at the end of this section. The following discussion describes the graphical data structure in the central computer.

Blocks in the structure

The graphic data structure contains four types of blocks: *node blocks, branch blocks, leaf blocks* and *data blocks*. These are shown symbolically in Figure 9. (Although a branch is represented by an arrow, it is not simply a pointer, but is a contiguous set of memory locations, as are each of the other block types.) Nodes

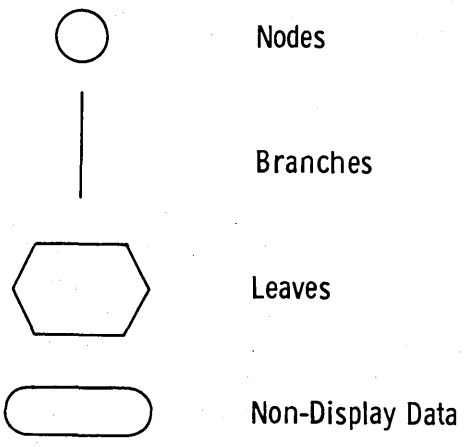


Figure 9—Blocks in the graphic data structure

and branches are blocks of fixed size; other blocks may be of arbitrary length. The generic form of graphical structures is the *directed graph* (with no closed loops). A directed graph is a set of *nodes* and connecting *branches* in which the branches have a direction as-

sociated with them, i.e., they point from one node to another. Some of the terminal nodes of the directed graph have special significance and are called *leaves,* in analogy with the way leaves are extremities on a tree. By convention, only leaves in the structure can specify displayable material. Other blocks are present only to provide structural information. Figure 10 shows the typical form of a graphical data structure.

Each node represents a particular sub-part of the picture. Each branch represents a particular occurrence, or *instance,* of the node to which it points. The branch may be thought of as a graphical call to a sub-part of a picture. It will cause the sub-part to appear at a position specified by information in the branch. Nodes serve to group together those sub-parts that the user may want to be associated with each other to form a larger sub-part. This nesting can continue to arbitrary depth.

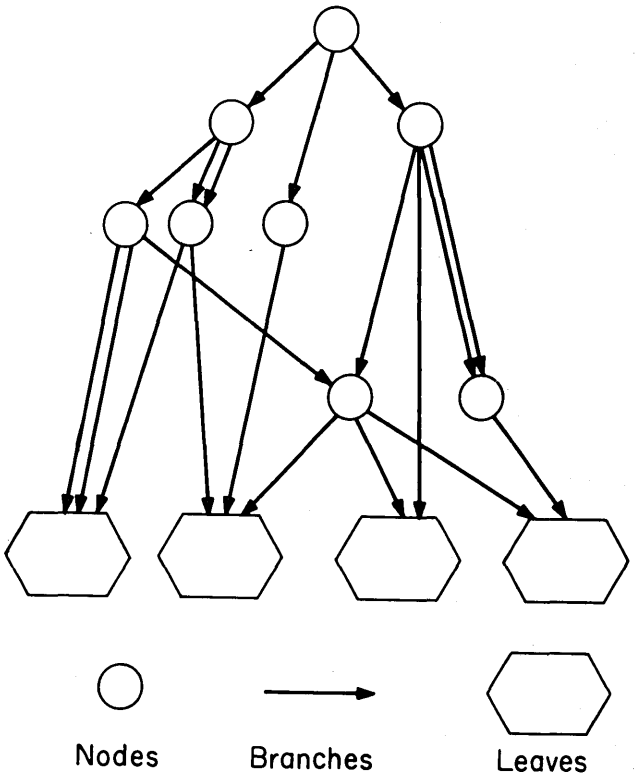
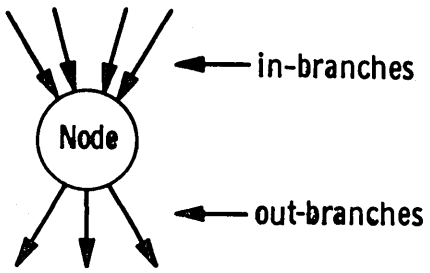


Figure 10—Typical graphic data structure

Nodes

Figure 11a shows the configuration of a typical node. Note that an arbitrary number of branches may enter a node, and an arbitrary number of branches may leave. It would not be possible to allocate a fixed size block for the node if pointers to all these branches were required in the node. To avoid this problem, the "in-branches" (branches entering the node) and "out-branches" (branches leaving the node) are organized



NODE BLOCK CONTENTS

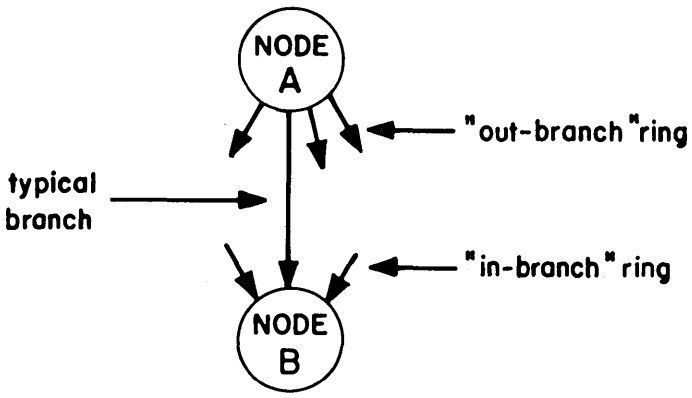
- Block type identifier
- Pointer to name data-block
- Pointer to first "in-branch"
- Pointer to last "in-branch"
- Pointer to first "out-branch"
- Pointer to last "out-branch"

Figure 11—Typical node-block and its contents

in separate "rings."[17] The node contains a pointer to the first branch in each of these rings, and each of these branches contains a pointer to the next branch, etc. Finally, the last branch contains a pointer back to the original node. A thread established by "back-pointers" also links all the elements in a ring in order to make tracing through rings faster for computer programs. Figure 11b shows the contents of a node block. In addition to pointers defining the "in" and "out" branch rings, each node contains a block type identifier (identifying it as a node) and a pointer to a data block that contains the symbolic name of the node. Thus every node (and therefore every picture subpart) may be referred to symbolically if desired. If the node has not been assigned a name, this pointer will contain a null value.

Branches

Figure 12a shows a typical branch connecting two nodes. Each branch has two rings passing through it: the "out-branch" ring associated with the node the branch starts on, and the "in-branch" ring associated with the node the branch terminates on. Figure 12b



BRANCH BLOCK CONTENTS

* Block type indentifier
* Name-data-block pointer
* "In-branch" ring pointers
* "Out-branch" ring pointers
* System non-display-data pointer
* User non-display-data pointer
* Branch x-displacement
* Branch y-displacement
* Branch display control parameters

Figure 12—Typical branch-block and its contents

lists the contents of a branch block. It contains a block type identifier (identifying it as a branch) and a pointer to a data block that contains its symbolic name. Thus every branch (and therefore every instance in a picture) may be referred to symbolically. If the branch has not been assigned a name, this pointer contains a null value. Next, the branch contains two ring elements. Each ring element contains three pointers: (1) a pointer in a forward chain linking all branches in the ring; (2) a pointer in a backward chain linking all branches in the ring; (3) a pointer to the node that is the "head" of this ring. These pointers facilitate rapid examination of the structure under program control.

Two pointers are included to data-blocks. These are the "system non-display data" pointer and the "user non-display data" pointer. If either of these pointers is not being used, a null pointer value is used for it. The system pointer and any data blocks linked to it are for the exclusive use of the graphical programming system. User programs may not access this information. The principal use of the system pointer is to point to a data block that identifies the branch as a light button. That data block contains the name of a program entry point that control is to be passed to if the instance represented by the branch is pointed at by a light-pen.

The purpose of the user non-display pointer is to allow the user (programmer) to attach non-display information to the graphic data structure. This non-display information is placed in "data-blocks," which are one of the four types of blocks found in the graphical data structure. These data-blocks are for the exclusive use of the programmer. They are defined and allocated under program control and may be of arbitrary format and size. They are intended to allow the user to specify information about instances that does not appear in the display.

Finally, the branch contains three fields necessary for the generation of the display itself. The X and Y displacements specify the distance between the display origins of the nodes (or node and leaf) which the branch joins. Since all display information in the structure is in terms of relative coordinate information (only the starting point of a display is an absolute coordinate), instances of sub-pictures can be moved about the display surface simply by changing the X and Y displacements in the proper branch. The last field contains display parameter information. Such things as intensity, scale and whether or not the light-pen is to be enabled during the display of the instance can be specified.

Leaves

Leaves specify how portions of a picture are actually to be drawn. They contain the "ink" that is visible in a portion of a picture. Data is placed in leaves by calls to data generating subroutines in GRAFCOM. Both text and line drawing information may be specified. When information is placed in leaves, the programmer is specifying information on a very large display surface. It is only when the information is finally displayed on one of the several available devices that a "window" is specified that establishes what portion of the display is to be visible on the output device. Data "grown" into leaves in real time by drawing on the GRAPHIC-2 scope must be rescaled before it is stored in the central data structure. Since a leaf may contain an arbitrary amount of data, its size cannot be specified in advance. For this reason, the leaf is broken into blocks of two types; the leaf-header block and leaf-data blocks. The leaf-header block contains a single ring-element, the "in-branch" ring, since by convention no branches point away from a leaf. It also contains a pointer to the first leaf-data block and to the last leaf-data block that make up the body of the leaf. These leaf-data blocks are linked by chain of pointers. A new leaf-data block is allocated and added to the chain whenever new data for a leaf are generated.

Example

The following example illustrates many of the features of the graphic data structure. Figure 13a shows a type of pattern familiar to anyone who has studied electronics. It is a diagram of a simple resistor-capacitor circuit. Because of generally accepted conventions, the meaning of this picture is immediately apparent to many people. Computers, however, have taken no electrical engineering courses. The pattern of lines might just as well be abstract art so far as the computer is concerned. The fact that a certain group of straight line segments symbolizes a resistor is not inherent in the picture, but is an organization imposed on the image by the viewer.

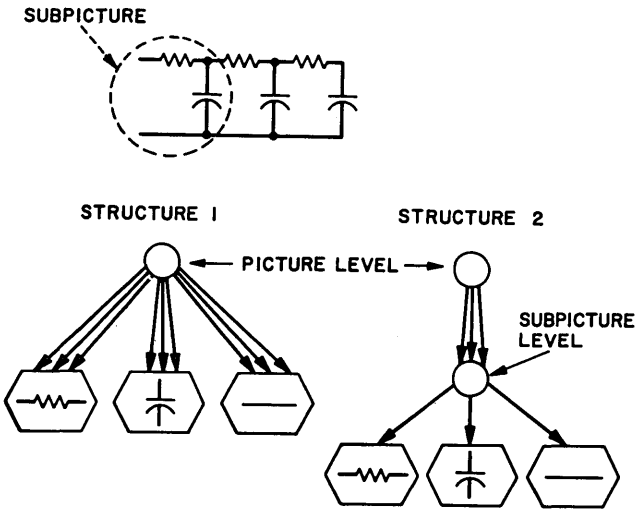


Figure 13—Simple circuit and two possible structures for it

A leaf is generated that contains the picture corresponding to each of the three types of elements in the circuit: resistors, capacitors and short-circuits. These elements may be positioned anywhere on the display surface by appropriate connecting branches. In the representation of Structure 1, shown in Figure 13b, there is no additional structure in the picture aside from the fact that each element type is a single leaf. In the representation of Structure 2 in Figure 13c, a sub-picture consisting of a resistor, capacitor and short circuit has been formed. In the whole picture there are three instances of this sub-picture (one for each of the branches pointing to the subpicture node), and one instance of each of the three basic elements.

Additional non-display data blocks would undoubtedly be linked to these branches in most real applications. They would contain such information as the electrical type of the elements being displayed and their values for use by analysis programs.

Modifications for GRAPHIC-2

The data structure in GRAPHIC-2 retains all the structural information present in the data structure in the central computer. GRAPHIC-2 is the only one of the display systems that uses information in structured form. All the others simply receive a set of display commands from the appropriate device Translator module. GRAPHIC-2 needs the structure because of the real time response it must give users at its console. It must be able to identify objects that are pointed at without requiring the intervenion of the central computer. It must be able to edit the structure, too. Thus, the GRAPHIC-2 data structure contains nodes, branches, leaves and data blocks that are in one-to-one correspondence with equivalent blocks in the central data base. Formats internal to these blocks are different, however. First of all, leaves in GRAPHIC-2 contain display commands in the format required to run the display scope, while picture information in the central computer is represented in a device independent way. Because a different dynamic storage allocation technique is used in GRAPHIC-2, a leaf is a single contiguous block of memory rather than a sequence of leaf-data blocks linked to a leaf-header. These leaves can be grown in real time under program control and may be of arbitrary length. Because space is a scarce resource in the small GRAPHIC-2 computer, an abbreviated pointer system is used to link blocks in the data structure. In particular, back pointers and pointers to the heads of rings are not used. Tracing one's way through the structure therefore may require more time, but time is a resource that is more readily available than space in GRAPHIC-2.

V. GRIN-2 language

The GRIN-2 (*GR*aphical *IN*teraction) language is a high-level graphical programming language that permits the generation and manipulation of the graphical data structure, and provides statements for controlling real-time man-machine interaction. The interaction portion of the language is used with the GRAPHIC-2 graphical terminal. The rest of the language pertains to the common data structure used by all graphical devices and terminals.

The following is not a specification of the GRIN-2 language, but a brief description of a few statements in several categories, the purpose of which is to give the reader the flavor of the language and allow an understanding of the programming example given in the Appendix. The categories described are:

(1) Real-Time Man-Machine Interaction
(2) Structure Generation
(3) Display Data Generation

(4) Structure Editing
(5) Display Control

Real time man-machine interaction

The basic man-machine interaction philosophy of the GRIN-2 language is one of the question and answer. The language provides statements which ask questions of the operator at a GRAPHIC-2 terminal. GRAPHIC-2 then waits (with control held by the language statement) until the operator answers the question by some light-pen or keyboard action. Control is then passed on, along with his answer, to other GRIN-2 statements which use the answer to direct further processing.

One of the real-time questions provided in GRIN-2 is asked by the WHICH statement. WHICH asks the operator the question *"which* of the objects displayed on the scope?" The operator then answers by pointing the light-pen at one of the objects. The meaning of this question depends on when it is asked. For instance, if a program to delete objects from the scope is in control it might use the WHICH statement to ask "WHICH of the objects displayed on the scope should be deleted?" When the operator pointed to one of the objects with the light-pen, control would pass on from the WHICH statement to the next statement. The next statements would then use the answer to delete the object picked.

The WHICH statement can also be used to direct the flow of control in the program through the use of light buttons. A light button is a displayable object which is associated with a program or subroutine. Given a number of light buttons displayed on the scope, the program in control could use the WHICH statement to ask "WHICH program should receive control next?" or "WHICH function should be performed next?" The operator could then pick the desired light button with the light-pen, and the WHICH subroutine would pass control to the picked light button's associated program.

Another basic real-time question in the GRIN-2 language is asked by the WHERE statement. WHERE asks the question *"where* on the scope?" by displaying a tracking cross which can be moved around the scope face with the light-pen. The operator answers this question by positioning the tracking cross with the light-pen. When the tracking cross is positioned where he wants it, the operator informs GRAPHIC-2 by pushing a button or taking some other action that can be specified as an argument in the WHERE statement. This action could even be picking a light button with the light-pen. Objects displayed on the scope can be made to move with the light-pen by "attaching" them to the tracking cross. Other GRIN-2 statements use WHICH and WHERE to form higher level statements such as DRAW. DRAW uses repeated calls to WHERE to allow the operator to draw a broken line segment on the scope.

Structure generation

Building a graphical data structure is done with three statements: LEAF, BRANCH, and NODE.

LEAF BLKPTR

generates an empty leaf-block and places a pointer to it in BLKPTR.

BRANCH (BLKPTR,FROM,DXY,PARM,TO,LB, DATA)

generates a branch-block and places a pointer to it in BLKPTR. FROM is a pointer to the node from which this branch is to descend. TO is a pointer to the leaf or node to which this branch is to point. DXY is a pointer to a DX, DY word pair which specifies the relative dsplacement on the scope between the display origins of the FROM and TO nodes. PARM is a pointer to a location which contains the parameters to be specified in the branch. If LB is given, it specifies that the branch is a light button and its associated subroutine is LB. DATA points to a user's data block that is to be attached to his branch. All of the arguments in the BRANCH statement are optional.

NODE BLKPTR ((B,DXY,P,T,LB,D) (. . .) . . .)

generates a node-block and places a pointer to it in BLKPTR. The next string of arguments specify a branch or set of branches which are generated and descend from the generated node. These arguments are similar to those in the BRANCH statement.

Display data generation

Display data in leaves are the scope commands which drive the scope and produce the picture. The following statements produce display data which is directed into the leaf specified by the last occurrence of either an INLEAF or LEAF statement (last generated leaf).

TEXT (CHI,CH2,CH3 . . .)

stores a series of characters in the last specified leaf. When the leaf is displayed, characters CH1,CH2, CH3 will appear on the scope.

VECTOR (DXY1,DXY2,DXY3 . . .)

generates a series of vectors. The arguments may be pairs of signed decimal integers or the location of a DX,DY word pair.

Structure editing

Statements in this category allow changes to be made to an already existing structure.

TCMOVE (BRANCH1,BRANCH2, . . .)

This is a dynamic statement which specifies that DX and DY (relative displacement) in the branches BRANCH1,BRANCH2 etc., are to be changed as the position of the tracking cross is changed in the next

WHERE statement. This allows objects to move with the tracking cross.

DETACH (BRANCH1,BRANCH2, . . .)

Specifies that the branches BRANCH1, BRANCH2, etc., should be detached from the structure.

ATTACH BRANCH,NODE

Attaches the branch specified by the pointer BRANCH to the node specified by the pointer NODE.

Display control

The structure or structures to be displayed are specified by attaching them to a special node called the DISPLAY NODE (DSPNOD). This may be done with the standard structure editing statements or by the special display control statements described below.

NEWDSP (BRANCH1,BRANCH2, . . .)

Any existing branches which are attached to the display node are detached and BRANCH1,BRANCH2, etc., are attached to the display node.

ADDDSP (BRANCH1,BRANCH2, . . .)

The branches BRANCH1,BRANCH2, etc., are attached to the display node without disturbing any branches which were already attached to it, adding them to the picture.

SUBDSP (BRANCH1,BRANCH2, . . .)

The branches BRANCH1,BRANCH2, etc., are removed from the display node without disturbing other attached branches (subtracting them from the picture).

Other categories of GRIN-2 statements include STRUCTURE TRACING, NON-DISPLAY STRUCTURE GENERATION, and SYSTEM CONTROL.

VI. Graphic-2 executive

The GRAPHIC-2 Executive is a set of programs that execute in the GRAPHIC-2's PDP-9 processor. It performs the functions of memory management, interrupt handling, data communication and display management. These functions are described below.

Interrupt handling

The interrupt handler is the heart of the executive program. All of the operator inputs to the GRAPHIC-2 (light-pen, pushbuttons, keyboard) are detected via interrupts as are conditions such as dataphone input and output, oscilloscope edge violation, and the occurrence of scope commands with an interrupt bit set. When an interrupt occurs, the trap handler gives control to the subroutine associated with that interrupt. This association is changeable from outside the trap handler, allowing programs to specify and change the reaction to any of the interrupts.

Data transmission

A 201B Dataphone (2,000 bits/sec) is used to transmit data back and forth between the GRAPHIC-2 and the GE-645. The data communication program controls the dataphone, buffers input and output data streams, detects errors and provides an error recovery strategy.

Memory management

The memory management programs give the PDP-9 a very large virtual memory, using the GE-645 and its file system for secondary storage. To make this possible the user's program, data and graphical data are broken up into blocks of variable size. Each block is given a unique 17-bit identification number (ID) before it is transmitted to the GRAPHIC-2, and all references between blocks are in terms of these ID's and an offset within the block. The ID is also used to request a block from the GE-645. To make interblock referencing practical, every block in the GRAPHIC-2 memory has assigned to it while and only while it is in core an entry in the BLOCK TABLE, which is located in the GRAPHIC-2 (see Figure 14). This entry establishes the correspondence between a block's ID and its location in the GRAPHIC-2, which makes possible easy relocation of blocks in core; only the core address in a block's entry in the block table need be updated when a block is moved.

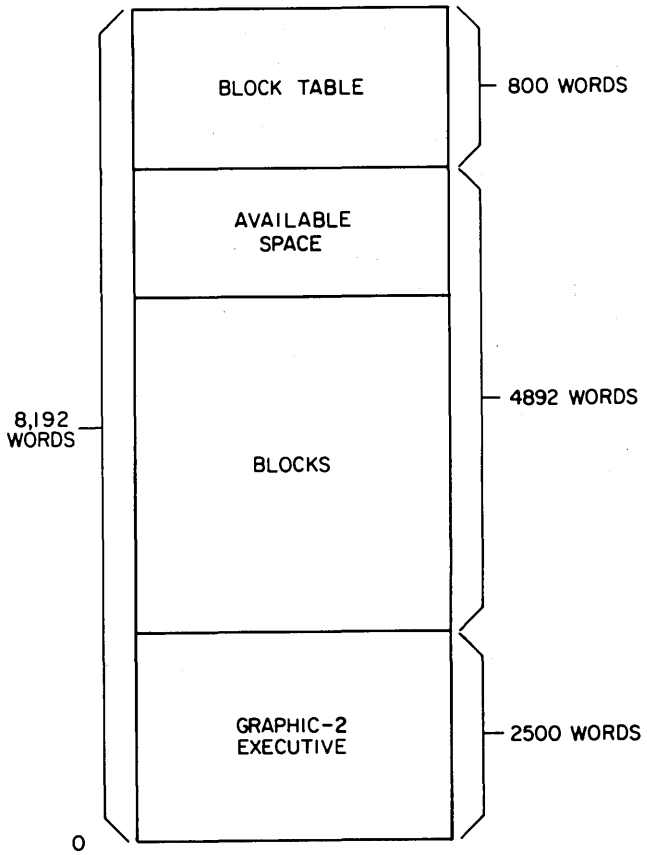


Figure 14—GRAPHIC-2 memory layout

The first time an inter-block reference is executed a search of the block table is made and the ID in the reference is replaced by its corresponding block table entry address (*BTEA*). Succeeding executions of that reference made before either of the blocks is removed from PDP-9 core, may be accomplished without having to search the block table. If the referenced ID is not in the block table then the block is not in core and a request for it is issued to the GE-645.

When a block is removed from core, its entry in the block table is also removed. At this time a search of every block in core is made to find any BTEA references to the removed block. These BTEA references are then charged to ID's. The hole created by removing a block is added to the available space (see Figure 14) by relocating all the blocks above it. The blocks are always contiguous in core.

In summary, when a block is not in core, all references to it are indicated by use of its ID When a reference is made to a block not in core, the block is fetched from the GE-645. An entry is made for the block in the block table and reference by ID is changed to a reference by BTEA. Other references to this block from any other blocks are *linked* (ID changed to BTEA) only when the reference is made. If the block is removed from core, all linked references are unlinked. The block table allows conversion from ID to BTEA (as references are linked) and vice-versa (when a block is removed). It also provides the starting core address of the referenced block so that an offset may be added to complete the reference.

Display management

To display the graphic data structure described in this paper, a semi-interpretive display program is used. A single leaf can be displayed by the scope without intervention by the PDP-9, but threading through the nodes and branches requires push down operations and help from the memory management programs. The display manager supplies this structure tracing function. It seeks out every path to every leaf in the specified structure and displays leaves as it encounters them. It uses the memory management program to link all references in the structure (if any are not yet linked) on the first pass through the structure. After the structure is linked, the interpretive and noninterpretive display functions are overlapped, i.e., while the GRAPHIC-2 scope-processor is displaying a leaf the display manager in the PDP-9 finds the next leaf and prepares to display it. Displaying interpretively has several advantages over a linked self-running display list. The relative positioning vectors in each of the branches, for example, are added up along the path to a leaf and the result put out as a single scope positioning command. This technique is much faster than executing all individual positioning vectors. Also, the pushdown list kept by the display interpreter contains a real time record of the display path. This record is very valuable when a light-pen strike occurs. Interpretive operation also allows the relative positioning vectors in the branches to be stored as full 18-bit Δx and 18-bit Δy values because scope command op-code bits are not necessary. This allows the GRAPHIC-2 data structure to contain an 18-bit by 18-bit picture, of which any 10-bit by 10-bit section can be displayed on the scope face.

VII. GE-645 graphic software system

Routines in the graphic programming system can be divided into two distinct categories: those which are accessible to a user by call or GRIN-2 language statement and those that are invisible to the user but provide necessary services. In the first category are sub-routines for building and editing the user's Graphic Data Structure and for creating and linking nondisplay information to it. In the second category are programs that handle communication between the GE-645 and all graphical devices, and the Translator programs that convert from internal GE-645 picture representation to formats required by specific devices. A dynamic storage allocator is used to allocate and free blocks that are used in the graphic data structure.

The most complicated device to handle from the point of view of the software system is the GRAPHIC-2. A library of GRAPHIC-2 subroutine blocks is maintained on GE-645 disk storage. These are kept in relocatable, linkable format since the GRAPHIC-2 executive system includes a linking loader that runs on the PDP-9. The equivalent routines in the GE-645 version are also kept on a library file. A critical element in the system is the "unique ID maintainer." In Section VI on the GRAPHIC-2 executive system it was pointed out that all program and data blocks in the GRAPHIC-2 have a 17-bit ID number associated with them. A complete dictionary of all blocks with assigned ID's is kept in the GE-645. This dictionary establishes a unique correspondence between blocks in or referred to in the GRAPHIC-2 and the equivalent block in the GE-645. Whenever GRAPHIC-2 needs a block of any kind it asks for it by ID number. The ID table must be used to locate the desired block in the GE-645. The block is then converted to GRAPHIC-2 format (if it is part of the Graphic Data Structure) or retrieved from the GRAPHIC-2 program file (if it is a program block) and sent over the communication link to the remote terminal. The unique ID assigned to a block is a dynamic operation that occurs during program execution and is completely invisible to the programmer.

The "real-time-input" simulator in the GE-645 is invoked wherever one of the real-time GRIN-2 statements is encountered in the GE-645 program. It simply supplies the next argument (or arguments) found on the input queue sent over from GRAPHIC-2. This allows the GE-645 version to undate its version of the Graphic Data Structure to correspond with events taking place at the remote terminal.

SUMMARY

A flexible graphics system that provides users with several different types of service has been described. Several devices of each available type are provided to insure good access to the system for many people. The system is intended for use in an environment where users do most of their own programming. Therefore, software support is provided that makes programming graphical I/O fairly simple.

A wide range of applications is expected. These include data plotting, circuit mask design and analysis, generating of motion pictures, text editing, and many more.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of many colleagues in the Information Processing Research and Machine Aids to Development Departments. In particular, we wish to thank A. D. Hause for valuable suggestions during the design of the system and L. Rosler for helping to crystallize many features of the GRIN-2 language.

APPENDIX

The following is a commented example of a graphically interactive program written in GRIN-2. This program allows an operator at a GRAPHIC-2 terminal to draw, move or delete broken line segments on the face of the scope with a light-pen.

```
 1 START      LEAF      DRAWLB
 2            TEXT      (D,R,A,W)
 3            LEAF      MOVELB
 4            TEXT      (M,O,V,E)
 5            LEAF      DLTLB
 6            TEXT      (D,E,L,E,T,E)
 7            NODE      LTBTNS( (B1,,,DRAWLB,DRAWPG)
                                (B2,(0,-30),MOVELB,MOVEPG)
                                (B3,(0,-60),,DLTLB,DLTPG)  )
 8            BRANCH    LBBRAN,,(950,950),,LTBTNS
 9            NODE      DRAWND
10            BRANCH    DRAWBR,,,,DRAWND
11            NEWDSP    (LBBRAN,DRAWBR)
12 HOME       WHICH     ,TRA
13            GOTO      HOME
14 DRAWPG     WHERE     WHICHX,DRAW1
15 DRAW1      LEAF      DRLEAF
16            BRANCH    DRBRAN,DRAWND,WHEREX,,DRLEAF
17            DRAW      WHEREX
18            GOTO      HOME
19 MOVEPG     WHICH     ,SAME
20            TCMOVE    WHICHB
21            WHERE     WHICHX,HOME
22 DLTPG      WHICH     ,SAME
23            DETACH    WHICHB
24            GOTO      HOME
```

Statement 1 generates a leaf in the data structure and a pointer to that leaf is placed in DRAWLB. The displayable text "DRAW" is then placed in the leaf (STATEMENT 2); statements 3,4,5 and 6 generate two more leaves with the text "MOVE" and "DELETE" in them. Statement 7 generates a node LTBTNS and three branches B1, B2, B3 (one connecting each of the three leaves to the node LTBTNS). These branches are also declared light buttons with the associated programs DRAWPG,MOVEPG and DLTPG. Different delta xy values are given in the three branches so that the three leaves (light-buttons) will be displayed at different locations on the scope. Statements 8, 9 and 10 then generate two more branches and a node. The total structure generated by statements 1 through 8 is then shown in Figure 15a. The node

DRAWND is a place to attach more structure when the DRAWPG light-button program gets control. Statement 11 specifies that the branches LBBRAN and DRAWBR and all the structure below them are to be displayed (Figure 15b). Control then is held in statement 12 asking the question "WHICH function should be performed next?" The TRA argument on the WHICH statement specifies that when a light-button is picked control should be given to its associated program. If a non-light-button were picked (impossible now since there are none), then control would pass to statement 13 which passes it back to 12 (WHICH). This in effect ignores any non-light-buttons and forces the operator to pick a light-button. Pointing to "DRAW" on the scope gives control to DRAWPG, "MOVE" to MOVEPG and "DELETE" to DLTPG. Assuming the operator picked "DRAW" with the light-pen, the first question asked by the drawing program DRAWPG is "WHERE should the broken line start?" The argument WHICHX in statement 14 is a pair of system locations which contain the scope coordinate of the light-pen strike that answered the last WHICH question. In this case this is the x,y location of where the "DRAW" light-button was touched. This intilizes the tracking cross to appear under the light-pen. The operator then moves the tracking cross to the place he wants to start drawing. Pushing a button on the keyboard causes control to pass to DRAW1 as specified by the second argument in statement 14. At this point the scope coordinates of the tracking cross are contained in a pair of system locations WHEREX. DRAW1 then generates a leaf DRLEAF and connects it to the node DRAWND with a branch DRBRAN. The initial position is also inserted in the branch with the WHEREX argument. The DRAW statement (17) then gets control along with the starting location WHEREX (answer from the WHERE question, statement 14). The opera-

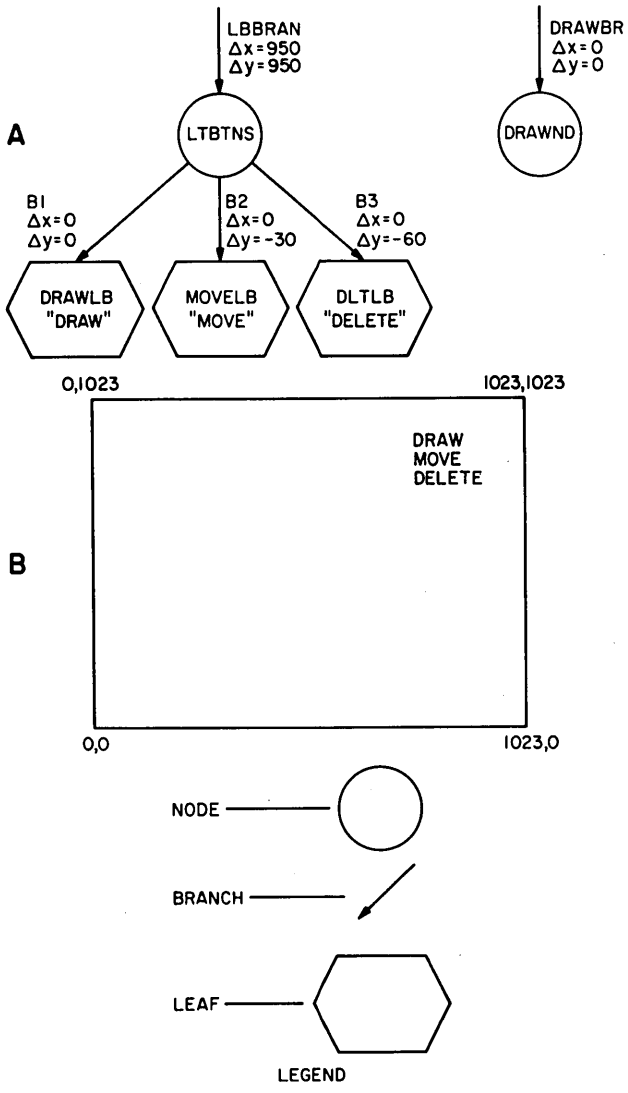


Figure 15—Structure and display for the GRIN-2 example (Appendix)

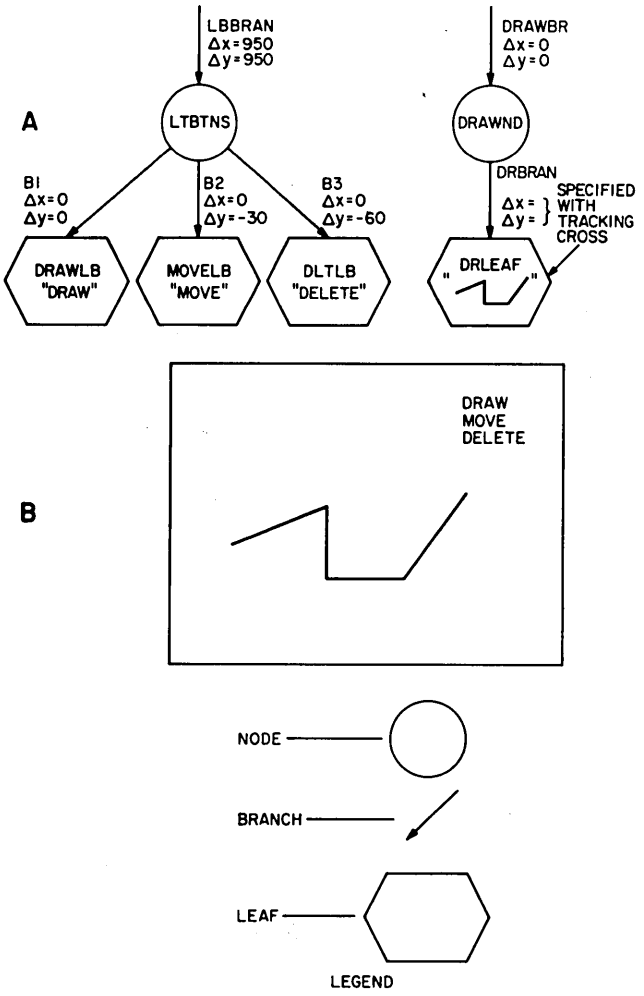


Figure 16—Structure and display for the GRIN-2 example (Appendix)

tor uses two buttons on the keyboard, one to indicate corners and one to indicate the end of the broken line segment. By repeated calls to WHERE and VECTOR, the DRAW statement allows the operator to draw on the scope. When he pushes the end-of-line button on the keyboard, control is passed to the next statement (18) which passes control back to the portion of the program which asks "WHICH function should be performed next?" Figure 16 shows the structure and picture at this point.

If the "MOVE" light button is picked with the light pen, MOVEPG will get control. This program then asks "WHICH branch should be moved?" (statement 19). The "SAME" argument specifies that light-buttons should be treated the same as non-light-buttons. In this case it means the light-buttons, as well as the non-light-buttons, can be moved around the scope. When the operator picks a branch to be moved, control passes to statement 20 which "attaches the object picked (answer left in WHICHB) to the tracking cross. The program then asks "WHERE should the branch be moved?" (statement 21). The picked object then moves with the tracking cross as it moves to follow the light-pen. The argument HOME specifies that control should pass to HOME when a button on the keyboard is pushed, indicating that the moved object is positioned to the satisfaction of the operator. The "DELETE" light button program should now be obvious.

REFERENCES

1 I E SUTHERLAND
Sketchpad: A man-machine graphical communication system
Proc. AFIPS Spring Joint Computer Conference, vol. 23, pp. 329-346 1963

2 E L JACKS
A laboratory for the study of graphical man-machine communication
Proc. AFIPS Fall Joint Computer Conference, vol. 26, pp. 343-350 1964

3 G J CULLER B FRIED
The TRW two-station on-line scientific computer
Computer Augmentation of Human Behavior

4 K C KNOWLTON
Computer produced movies
System Analysis by Digital Computer, edited by F. F. Kuo and J. F. Kaiser, John Wiley and Sons, New York, Chapter 11, pp. 375-394 1966

5 F H HARLOW J P SHANNON J E WELCH
Liquid waves by computer
Science, vol. 149, no. 3688 p. 1092 September 3, 1965

6 H C SO
Analysis and iterative design of networks using on-line simulation
System Analysis by Digital Computer, edited by F. F. Kuo and J. F. Kaiser, John Wiley and Sons, New York, chapter 2, pp. 34-58 1966

7 W R SUTHERLAND
The on-line graphical specification of computer procedures
Lincoln Laboratory Technical Report no. 405, Lexington, Massachusetts 1966

8 W H NINKE
GRAPHIC-1: A remote graphical display console system
Proc. AFIPS Fall Joint Computer Conference vol. 27, pp. 834-846 1965

9 F J CORBATO V A VYSSOTSKY
Introduction and overview of the multics system
Proc. AFIPS Fall Joint Computer Conference, vol 27, pp. 185-196 1965

10 E L GLASER
System design of a computer for time-sharing applications
Proc. AFIPS Fall Joint Computer Conference, vol. 27, pp. 197-202 1965

11 V A VYSSOTSKY F J CORBATO R M GRAHAM
Structure of the multics supervisor
Proc. AFIPS Fall Joint Computer Conference, vol. 27, pp. 203-212 1965

12 R C DALEY P G NEUMANN
A general-purpose file system for secondary storage
Proc. AFIPS Fall Joint Computer Conference, vol. 27, pp. 203-212 1965

13 J F OSSANNA L MIKUS S D DUNTEN
Communications and input-output switching in a multiplex computing system
Proc. AFIPS Fall Joint Computer Conference, vol. 27, pp. 213-230 1965

14 H S McDONALD W H NINKE D R WELLER
A direct-view CRT console for remote computing
Digest of Technical Papers, 1967 International Solid-State Circuits Conference, vol. 10, pp. 68-69.

15 W H NINKE
A satellite display console system for a time-shared computer
In Preparation

16 D T ROSS J E RODRIGUEZ
Theoretical foundations for the computer-aided design system
Proc. AFIPS Spring Joint Computer Conference, vol. 23, pp. 305-322 1963

17 L G ROBERTS
Graphical communication and control languages
Proc. Information Systems Sciences Second Congress, pp. 211-217 1964

Reactive displays: improving man-machine graphical communication

by JOHN D. JOYCE and MARILYN J. CIANCIOLO
Research Laboratories
General Motors Corporation
Warren, Michigan

INTRODUCTION

The on-line graphic representation and solution of problems is opening the door to new and exciting computer applications. Continuous man-machine interaction via graphic consoles makes feasible the solution of entirely new classes of problems. This expanding use of computer graphics is requiring improved techniques of man-machine communication and graphic data management. At the General Motors Research Laboratories, we have had the opportunity since 1962 for considerable experimentation in a man-machine environment. From these experiments new ideas have evolved about how to improve the two-way information flow between the console user and the computer model of his problem. A fundamental concept is the reactive display, which supplies immediate graphical response to the actions of a man at a console. We have found that reactive displays provide a good basis for interaction between the man and the individual phases of his problem.

A problem to be solved is that of providing a good foundation in graphic displays. The characteristics of this foundation must be such that console users can easily do productive work at a console for several hours per day. These console users in general will have no computer training and will have no inclination or desire to become encumbered by the intricacies of conventional computer work. These users may be designers, draftsmen, electrical engineers, or project planners who merely want to communicate with a console in order to get a job done. Our experiences have demonstrated that using alphanumeric language statements to produce graphic displays is not the best environment for the types of console users mentioned above. An alphanumeric language is an abstraction of a problem which is harder to understand than a pictorial abstraction of a problem. For example, a project planner could describe all the activities of a project, their durations and the precedence relationships which exist among the activities, solely with an alphanumeric language. However, this same information is much better described with a PERT diagram.

Let us consider some of the steps involved in using a computer to help solve a problem. For a segment of a problem, a sequence of one or more procedures is requested, several data items are entered as inputs to the procedures and results are produced. Certain sets of procedures have specific relationships. Only certain sets of data items satisfy requirements for particular procedures. All other data items can be considered to be background or contextual information when a particular procedure is in control. A foundation for graphic displays must provide the facilities for representing and allowing the selection of individual procedures, actions, and data items pictorially. It is also essential that the graphic system dynamically represent changing relationships among various sets of procedures and data items. Many errors can be eliminated by only allowing selection of syntactically correct inputs and by providing dynamic feedback to ease the correction of other human errors.

Let us assume that a project planner wants activity A to have a longer duration, a different description, and different precedence relationships; also, that a PERT diagram is displayed on the console and that a "Change Activity" procedure is in control. All precedence lines are made non-selectable by a light-pen. However, all precedence lines remain displayed so that the activities are viewed in context with the precedence lines. The user places the light-pen on the screen. Activity B becomes brighter. Although this selection is syntactically correct, the user has made a human error and adjusts the light-pen position until Activity A becomes brighter. Satisfied with the immediate feedback the user removes the pen from the

screen and now a different set of items can be changed, i.e., (a) the description of A, (b) the duration of A, and (c) the lines of precedence of A. And so the process continues with different sets of display entities changing to reflect the ever-changing status of the problem.

Objectives

Three aspects of a graphic system will be presented. (See Figure 1.) First, we shall discuss some of the human factors of computer graphics, i.e., what is the best problem-solving environment for the man at a console. Second, we shall describe in detail the graphic programming system. They key elements of this system, its internal structure, and the advantages and disadvantages of certain implementation techniques will be presented. By graphic programming systems we mean the software support which is the linkage between the display hardware and the applications programmer. Third, we shall consider the facilities as seen by the applications programmer. It is his job to build upon the graphic programming system a graphic language which will best communicate with the console user.

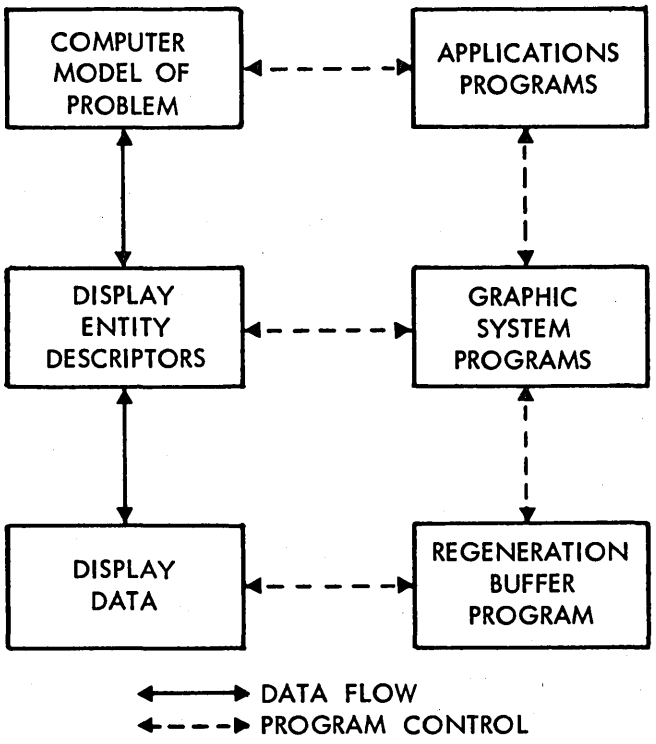


Figure 1 – Graphic system programs as an interface

In this paper we shall limit ourselves to a discussion of the tools we have designed and implemented to interface between the applications programmer and the graphic console user. We shall make no attempt to discuss the design or implementation of a graphical language. Many kinds of man-machine communications media could be built atop the graphical system we discuss in the following pages.

Definition of terms

A few terms will be defined here for clarity. These definitions are not suggested as general definitions but only for their use in the remainder of the paper.

A *user* or *console user* will refer to a person who manipulates console controls such as a light-pen function buttons, typewriter, etc., to do productive functions such as mechanical or electrical design or parts layout. This person usually has no knowledge of computer hardware or software and is only interested in the console as a tool to get his own job done.

An *applications programmer* shall mean the person who provides the software to do specific productive jobs. This software is built upon the graphic system software and the time-sharing system software.

A *graphical language* is considered to be one in which communication at the CRT is carried out by continuous interaction with pictorial representations of a problem. In such a language very little use is made of alphanumerics, except for labels and a few key control words. A language which generates pictures from executing groups of alphanumeric statements is not considered to be a graphical language.

The *human factors* of graphics refers to all aspects of communication with a man at a console. The object here is to provide the maximum comfort and convenience of operation for a console user and to increase efficiency by keeping good response time and eliminating obvious sources of errors of syntax. These points will be discussed in more detail under the section on Man-Machine Communication Techniques.

Man-machine communication techniques

No graphical communication system will be an effective problem-solving tool unless it is natural and convenient to use. Consequently, at the GM Research Laboratories we have given a great deal of attention to optimizing user satisfaction. The fundamental goal has been to bring the man ever closer to direct communication with a computer. This communication should be in a conversational graphical language which the man understands.

Discrete problem elements and procedures

The beginnings were primitive: a display was thought of as one large picture, a kind of visual feedback. Language statements were fed into a card reader by the user and executed. The results were displayed on the console screen. Gradually, console use expanded to communication more directly in graphics.

It was not evident in the beginning that working toward a solution of a graphical problem involved the manipulation of individual data elements. For example, a mechanical parts designer will work with an individual line when he is changing the design. An electronic circuit designer will change a circuit by working with an individual component and its relationship to other components.

Focusing the user's attention

Another aspect of external communication was not apparent during initial attempts at a graphic system. This is the need to gain the attention of the console user by directing him to a certain portion or portions of the display. There are many ways of achieving this: possible methods are intensification, blinking, modulation of size, or changing the color of portions of the display.

After investigating several possibilities, we concentrated on the use of intensification for highlighting parts of a display. For example, the four boundary lines of a surface may be brightened while the interior lines are dimmed. This technique of varying intensification has proven to be an important piece of feedback to the console user. We call the preceding example "static" intensification because an item remains intensified throughout the current display. Static intensification can be used meaningfully in other ways, besides highlighting. For example, user attention can be focused on suggested choices of control words by brightening them, while dimming the rest of the display. Shading to give a three-dimensional effect is also useful in optimizing the user's understanding of a display. This technique, however, requires enough intensity levels so that the transition between them is continuous and imperceptible.

Dynamic intensification is a more important form of brightening. An item, a line and its label for example, is displayed at a very high level of brightness during the time that the pen is pointing at it. (See Figure 2). Whenever the pen moves to a void area, nothing is intensified. If the pen is pointed at another part of the display, that part will be intensified. Intensification of the selected item is feedback that tells the console operator exactly what he is pointing at. When the operator is satisfied with his selection, he removes his pen from the screen.

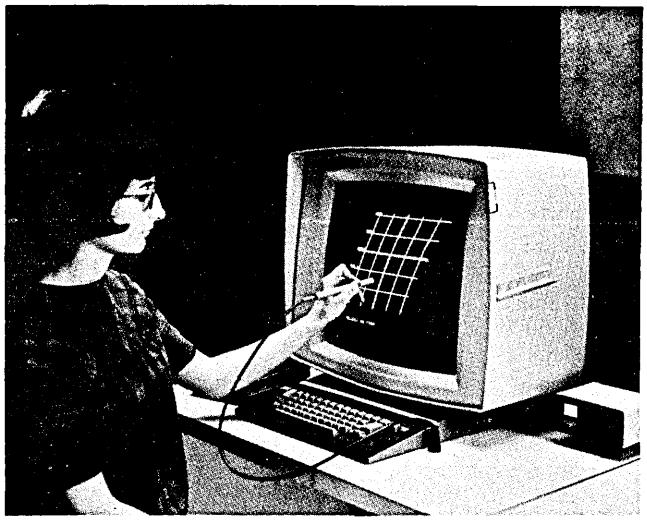


Figure 2 – Dynamic intensification of a display entity

Restricting selectability of display items

Another important mechanism in the restricting of selectability to only certain items or classes of items in the display. It is obvious that some background information such as borders or grid lines may be displayed, but is never selectable. Perhaps not so obvious is that, frequently in reactive displays, classes of items should be made non-selectable. The console user might, for example, wish to execute a line smoothing procedure. He will probably be required to indicate the line in which he is interested. At this point in time, then, only lines should be candidates for selection. All other kinds of items should be "disabled" for detection. Further, only lines should be giving a feedback response in the form of dynamic intensification.

This "selective disabling" aids the operator in recognizing his displays as a composition of various sets of items. In addition, he will be less confused when trying to make a selection. Only a small subset of items can be selected, even though other information is still displayed. Thus, the selectable items are kept is context with the rest of the display. The combination of selective disabling and dynamic intensification provides the means of conveying the syntax of a graphical language to the console user.

Other techniques

There are other human factors involved in console use, which are not a direct concern of the graphical systems designer. They merit attention because these external criteria must be met with a minimum of effort by the graphical language implementer.

In production design work, for example, console users are sometimes at work for several hours or more. This prolonged use clearly indicates the need to minimize the number of manipulations required to solve a problem, e.g., the number of "pokes" at the screen. It is the job of a good graphical system to interface with the graphical language in meeting this important requirement.

The graphical language can further enhance user understanding by providing the ability to

(a) change the scale or size of the data,
(b) change the direction of view,
(c) rotate or otherwise transform data,
(d) label items on the screen.

When the user sees these actions carried out in continuous motion, under his control, his visualization of the problem is improved. A good graphical system lays the groundwork for implementing these capabilities. It does this by providing the means of effortless communication between the applications programmer and the graphic device.

All the graphical techniques just described help to improve the all important information flow, the conversation, between the computer and the man. Each small part of the display now becomes a variable under the control of the man. He can follow the progress of his problem and analyze intermediate results. He can, in fact, change the flow of activity to an entirely new direction. He is continuously reacting to the everchanging display, and equally important, the display is reacting to him.

A characteristic of the above graphical communication techniques is the high burst rate of information flow. A console user can receive information with his eyes at rates as high as 4.3×10^6 bits/second.[1] He can transmit decisions back to the computer at rates as high as 10^2 bits/second. These high rates, although they cannot be sustained, do require an internal data structure which provides efficient display handling and good response time.

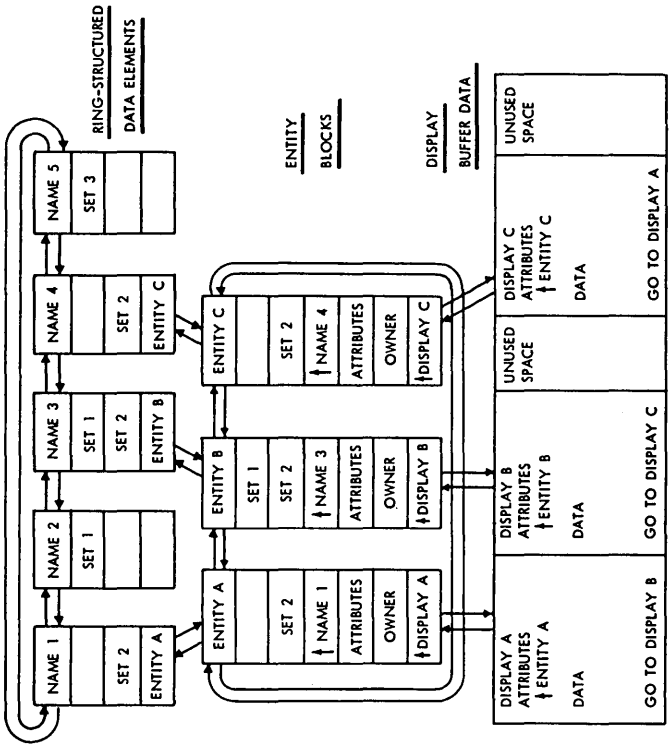


Figure 3 – Elements of reactive displays

Graphic programming system

Elements of a reactive display

The preceding paragraphs have described some of the external requirements met by the graphic programming system at the GM Research Laboratories. It is now appropriate to consider the basic internal structures that can help meet these needs most efficiently. (See Figure 3.)

As mentioned earlier, a display can be thought of as one large picture. This means display changes require working with the entire display. Another approach is to change a display by manipulating only a small part of it. This involves a decomposition of the large display into smaller displays called "entities." An entity may be defined as a distinct collection of displayable data to which the user may wish to make a unique reference. It represents some meaningful part of the model of the problem. The display data comprising an entity may be any combination of vectors, characters, or points. It may be scattered anywhere on the screen.

Each entity has display attributes assigned to it, which determine its external characteristics. It is described by a level of beam intensity and the attribute of selectability or non-selectability. Further, to

speed internal manipulation, each entity may be classified by its membership in a set(s). The membership of each entity in zero or more sets determines its relationship to the other entities in the display. In addition, the relations between display entities reflect the relations of the data items to which they correspond. To further aid in rapid data management, a two-way direct link has also been established between every displayed data item or control word and its display buffer counterpart.

The use of these basic structural ideas has helped to create a graphic system which can quickly and effectively handle the flow and manipulation of graphical data. The system affords a natural means of communication with the console user and, at the same time, ease and flexibility to the applications programmer.

Structure for entity descriptors

The need for flexible graphic control has impressed upon us the importance of certain internal requirements. No matter what type of display system is being implemented, it is useful to have blocks of entity descriptors, each of which describes an entity and its properties. Blocks in current use are linked together in a ring structure. (See Figure 3.) In the same way, unused blocks are linked together in a "free space" ring. They can be obtained quickly when another entity is to be added. Each active block contains a pointer to the previous entity and to the next entity, information about the entity attributes, and the size and display buffer location of the entity data. (See Figure 3.) It is assumed that the display data itself resides in the display buffer memory. The order of the entity descriptor ring matches the order of regeneration in the display buffer. This implicit information is used when adding or deleting entities.

It might be argued that a separate ring structure to describe display entities is unnecessary. Display characteristics could be included in a ring structure which described the data elements themselves. However, our experiments have indicated that a great deal of CPU processing time, as well as time to access peripheral storage devices,* was required to search the data ring structures. A less time-consuming method must be used to manipulate display entities since online response time is a major consideration in graphical work.

*To minimize the amount of CPU memory used most of the large data structure was maintained on high-speed drums. Searching a ring structure which threaded through the data often involved bringing many "pages" of data into CPU memory. Access time to a given data item was a function of the depth at which it was buried in the ring structure.

Data elements which correspond to display entities are generally a small subset of those which describe the entire problem. Thus a small amount of memory can contain a ring structure that describes the display characteristics of displayed data elements and the displayed control words, which have no data element counterpart.

The compactness of entity descriptor information offers advantages in both a paging and non-paging environment.[2,3] With paging and ring structures the descriptor blocks can be contained within one or two data pages. If it is assumed that pages will remain in memory based on frequency of use, then the blocks will nearly always be accessible without retrieval from a peripheral device. If the blocks are "paged out" while the console user is thinking, their compactness will permit rapid retrieval. The blocks will remain in memory so long as frequency of graphic attentions is high.

In the non-paging environment the small size of the entity descriptor ring makes it possible to keep it in the CPU. Many additions and deletions of blocks can therefore be carried out quickly and thus, maintain good response time for the console operator.

Control of execution sequences

It is important to give to the applications programmer the ability to enable or to inhibit the use of hardware features. These might include a light-pen, an alphanumeric keyboard, or programmed function keys. A single interrupt handling code in the graphics system dispatches control to the appropriate place when these features are activated. Transfer of control is carried out by having the user supply an "owner" (in the form of a program name or statement label) for each hardware feature and each display entity. The owner is recorded in an appropriate control block. When an enabled feature or entity is selected, control is transferred to the corresponding "owner." This eliminates the need for applications programs to decode attention information.

Linking display entities and problem elements

Problems of search and table look-up are eliminated by a direct two-way link between a display entity and the data element to which it corresponds. The applications program assigns to each entity a name or pointer which either has a distinct meaning in the computer model of the problem, or is a direct pointer to its associated data element. A graphics system program provides another pointer which is guaranteed to be unique. This pointer is directly linked to an entity for subsequent changes to or deletion of that

entity. When an entity is selected, both pointers are returned to the owner program. Experiments have shown that a direct two-way link between a display entity and a data element maximizes efficiency in manipulating the display.

Display hardware capabilities

In order to make the two-way pointer system work most effectively, the display controller hardware should have certain logical capabilities. Without CPU intervention the controller should be able to "name" every new entity as the pen passes over that entity. It can do this by physically moving the name or label of the detected entity to a fixed buffer location. When the final selection is indicated by the man removing his pen from the screen, the graphics system program needs only to read the contents of that fixed buffer location. The pointer to the corresponding data element or control word is thus available without any further processing. To make the concepts of reactive displays most effective in a time-sharing environment, it is desirable to reduce the number of CPU interrupts to a minimum.

We found this could be best accomplished by making additional demands of the graphic hardware. Specifically, we needed the ability to perform dynamic intensification of display entities without CPU intervention. We also needed to handle display attributes from within the controller. This required hardware control of display intensity and the enable-disable status of entities.

These capabilities imply that the display controller can:

(a) detect light and remember this information,
(b) do a conditional transfer on the remembered detection,
(c) move the name of the detected entity to a specified location,
(d) control beam intensity with a single buffer order,
(e) enable or disable graphical data with a single buffer order.

(See Figure 4.)

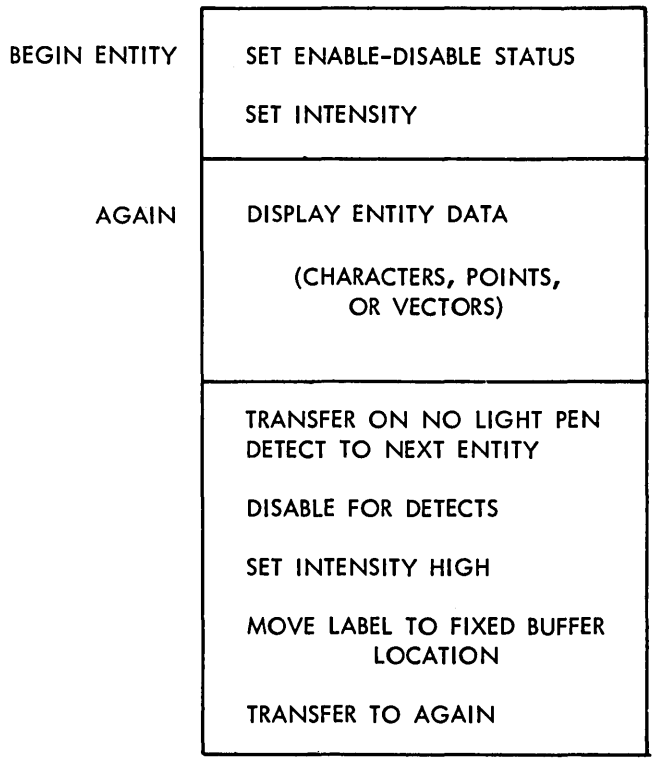


Figure 4—Entity structure in display buffer

Associative relationships among entities

A graphical display is simply a visual representation which helps the console user to understand a computer model of his problem. Within this model relationships exist between one data item and other data items. In a job scheduling model, for example, a given job is related to all the jobs which must be completed before that given job is begun. In the model of a design problem two points are related by a distance. Another kind of relationship exists in which one data element is related to others because they are members of the same set. For example, all lines in a drawing belong to one set, while all lines which belong to Surface A comprise a sub-set of the original set.

The need to manipulate data items based on associative relationships is illustrated by the fact that whole new languages have been written expressly for that purpose.[4,5] We have found that in manipulating graphical display entities, it is at least necessary to work with sets of entities or logical combinations of sets of entities. Suppose, for example, that at some point in time only a "line entity" which lies on "Surface A" is an acceptable screen selection. The entities in this set need to be rapidly enabled for operator selection, while all others should be disabled. At some other point in time all "point entities" on a "Centerline" should be brightened, or only the control words "LINE SMOOTHING" or "SURFACE EVALUATION" may be applicable. The display system must be able to react rapidly to such requested changes in display characteristics.

From the above examples we see that the relationships among display entities are simply a reflection of the relationships which exist among their corresponding data items. We have found it sufficient to record

the set membership of entities in the entity descriptor blocks. Tests on an experimental system indicated that the assignment of each entity to membership in one or more sets was convenient and adequate to handle display manipulations based on associative properties.

Handling the associative relationships among displayed items on an entity basis means that the graphical system does not need to work with the entire data structure. Since the data set for the entity descriptors is almost always smaller than the total data set for the model of a problem, there is a real advantage to the above method. In addition, display characteristics are tied directly to the associative relationships. Display changes are thus speeded up because the same program which searches for the requested set of entities can change the appropriate display characteristics at the same time. In a data environment where the associative relationships among data items are not considered at all, handling sets of display entities at the graphical system level provides a capability not otherwise possible.

The applications programs supply the names of the sets (if any) of which each entity is to be a member. The graphic system saves these names and the information about the set membership of each entity. As additional entities become members of existing sets, they are marked as such. A new set is created whenever an entity has membership in one or more sets which are currently unknown to the graphic system. Sets may become empty when entities are deleted, or whenever set membership of one or more entities has changed. Empty sets are automatically purged from the system when the additional space is required for new sets.

An incident matrix was selected to record the membership of entities in various sets. The expected maximum number of entities (150-200), the maximum number of sets (25-30), and the distribution of entities among sets indicated that an incident matrix would conserve storage and computer time. One row of the matrix is stored with each entity descriptor.

The disadvantage of this method is that it is difficult to accommodate more than an arbitrary number of sets at a given time. A ring structure could be used to describe associative relationships, but would require more computer time and storage.

Logical combinations of sets of entities may be deleted or have their properties changed by one request from the applications program. Any combination of sets which can be expressed by "AND," "OR," and "NOT" operations can be handled. Some of the more frequently used combinations are of the following types:

(a) entities which are members of Set A "OR" Set B,
(b) entities which are members of Set A "AND" Set B,
(c) entities which are members of Set A "OR" Set B and "NOT" of Set C.

These unlimited combinations of sets have provided facilities for making display changes conveniently and efficiently. The ability to make such changes provides a syntactical basis for graphical displays.

Display buffer management

No system restrictions have been placed on either the size or number of display entities which can exist at one time. The physical size of the regeneration buffer provides an upper limit. When an entity is deleted from the display buffer, the space it occupied is given back to the graphic system. The regeneration cycle is altered only to the extent that pertinent transfer addresses are changed. The entity previous to the deleted entity now transfers to the entity which followed the deleted entity. This method of deletion has several advantages:

First, there is no need for the system to maintain a copy of the regeneration data in CPU memory. Second, no processing time is required to recreate the regeneration cycle and store it as a contiguous block in buffer memory. Third, the amount of data transferred to the display buffer is minimal.

To summarize, entity deletion as described above saves CPU memory space, CPU processing time, and I/O time. One disadvantage of this method is that variable length blocks of unused space remain in the buffer. Since these holes may or may not be large enough to accommodate a new entity, efficiency of space usage becomes an item of concern.

A graphic system using variable length entities and this form of deletion was written and sampled to test fragmentation of the buffer memory. The results indicated that only a moderate percentage of space was wasted on holes interspersed among entities. Extensive testing showed that the number of holes did not usually exceed 10% of the maximum number of entities. Even with complex applications, the number of entities does not normally exceed 150-200 within the framework of current CRT sizes and regeneration buffers. This means that a small CPU memory table can accommodate all the information needed to manage buffer space allocation.

Facilities for applications programmers

The capabilities of the graphic system described in this paper are available to the applications pro-

grammer for a high-level language (PL/I). The following can be easily accomplished:

(a) Creation of new display entities
(b) Deletion of entities
(c) Modification of entity properties
(d) Modification of entity data
(e) Modification of properties of logical combinations of sets of entities
(f) Deletion of logical combinations of sets of entities
(g) Enabling of hardware functions (typewriter keyboard, function buttons, light-pen for positional input)
(h) Temporary inhibition of all interrupts and re-enabling for interrupts again

Each one of the functions listed above is available by an individual subroutine call in PL/I. The programmer provides only X,Y data and attribute information when he creates an entity. He need not be concerned about the management of space in the display buffer. All display control orders and logic are generated automatically by a graphic systems routine. I/O generation and transmission are also done automatically.

When the graphic system has information to present to the problem program, CPU control is transferred to the appropriate "owner" location. This location was previously indicated to the system for each display entity and hardware function. The system also makes available information about the interrupt type and, if applicable, the X,Y position of the pen.

The applications programmer is thus freed from the task of interfacing his computer model of a problem with the display hardware. The two-way information flow is carried out quickly and efficiently at a systems level. The result is good response time and economical CPU operation.

SUMMARY

The principles of reactive displays can be used in many different computer console arrangements. At General Motors Research Laboratories three graphic systems have been written, all of which utilize the concepts described in this paper. (See Appendix.) In spite of internal dissimilarities and hardware differences, these three systems provided nearly identical capabilities to the applications programmer.

The reactions of both the applications programmers and console users have been enthusiastic. The applications programmer appreciates the flexibility and efficiency of the system. He can easily communicate with the graphic hardware and has access to specific information about user respones. Console users indicate that they especially like the dynamic intensification of entities and fast response time. They also appreciate the selective enabling of only meaningful entities. From the standpoint of internal efficiency, our current thinking is that graphic systems based on concepts of reactive displays offer maximum speed and ease of data manipulation.

Appendix

Three graphical systems using the principles discussed in this paper have been implemented at the General Motors Research Laboratories. The main features in these systems are listed in Table I.

	SYSTEM A	SYSTEM B	SYSTEM C
CPU	IBM 7094	IBM 360/50	IBM 360/67
LANGUAGE	NOMAD	PL/I	PL/I
SOFTWARE ENVIRONMENT	MULTIPROGRAMMING	TIME-SHARING	TIME-SHARING
DISPLAY CONSOLE	DAC-I CONSOLE	IBM 2250-I	IBM 2250-III
REGENERATION BUFFER	WITHIN CPU	ONE PER CONSOLE	SHARED BUFFER (IBM 2840-II)
DISPLAY CONTROLLER LOGIC	ALMOST NONE	LIMITED AMOUNT	MODERATE AMOUNT
POINTING DEVICE	VOLTAGE PENCIL	LIGHT PEN	LIGHT PEN

Table I—System configurations

The first system was built upon the software and hardware of the DAC-I System.[6,7,8] The IBM 2250-I consoles used in the second system were used for program checkout until the IBM 2250-III consoles and 2840-II controller were installed. This latter combination of hardware included, in addition to the Graphic Design Feature, orders for a conditional transfer on light-pen tip switch open and orders to control display intensity.

ACKNOWLEDGMENT

The authors wish to express their appreciation to Edwin L. Jacks and Fred N. Krull for their help in the development of the ideas presented in this paper.

REFERENCES

1 H. JACOBSON
The informational capacity of the human eye
Science, vol. 113, b, pp. 292-293 1951

2 J G DENNIS
Segmentation and the design of multiprogrammed computer systems

J. of Association for Computing Machinery vol 12 no 4 pp 589-602 October 1965

3 B W ARDEN B A GALLER T C O'BRIEN F H WESTERVELT
Program and addressing structure in a time sharing environment
J. of Association for Computing Machinery vol 13 no 1 pp 1-16 January 1966

4 G G DODD
APL-a language for associative data handling in PL/1
1966 Proc FJCC vol 28 pp 677-684

5 L G ROBERTS
Graphical communication and control languages
Second Congress on the Information System Science Spartan Books Washington D C 1964

6 E L JACKS
A laboratory for the study of graphical man-machine communication
1964 Proc FJCC vol 26 pp 343-350

7 B HARGREAVES J D JOYCE G L COLE et al
Image processing hardware for a man-machine graphical communication system
1964 Proc FJCC vol 26 pp 363-386

8 M P COLE P H DORN C R LEWIS
Operational software in a disc-oriented system
1964 Proc FJCC vol 26 pp 351-362

BIBLIOGRAPHY

1 S H CHASEN
The introduction of man-computer graphics into the aerospace industry
1965 Proc FJCC vol 27 pp 883-891

2 T E JOHNSON
Sketchpad III: A computer program for drawing in three dimensions
1963 Proc SJCC vol 23 pp 347-353

3 W H NINKE
Graphic 1-A remote graphical display console system
1965 Proc FJCC vol 27 pp 839-846

4 M D PRINCE
Man-computer graphics for computer-aided design
Proc. of IEEE vol 54 no 12 pp 1698-1708
December 1966

5 D E RIPPY D E HUMPHRIES
MAGIC-A machine for automatic graphics interface to a computer
1964 FJCC vol 27 pp 819-830

6 R STOTZ
Man-machine console facilities for computer aided design
1963 Proc SJCC vol 23 pp 323-328

7 I E SUTHERLAND
Sketchpad: A man-machine graphic communication system
1963 Proc SJCC vol 23 pp 329-346

Graphic language translation with a language independent processor

by RONALD A. MORRISON
General Electric Company
Cincinnati, Ohio

INTRODUCTION

Since January 1963 when "Sketch Pad, A Man-Machine Graphical Communication System"[1] by Ivan E. Sutherland was published as a Ph.D. Thesis at M.I.T., graphic displays on computers have been used for a variety of experimental[2,3] and production[4] purposes. The motivation behind the system described here, is to provide an interactive graphic input/output facility to a number of *existing* software systems. Our business, like many others, has developed over the past 10 years a number of sophisticated and complex "Computer Aided Design" systems. These systems include principally numerical control part programming systems and engineering design analysis systems. Although these systems have proven themselves to be a *necessary* part of the design, and manufacturing process, nevertheless preparing input and assimilating output is still a time consuming process for the users. The interactive display speeds up these processes. Its output is more meaningful, too, since pictures as well as numbers and text are used as communication media between the man and the machine.

The term "graphic language" has been used ambiguously, in the literature, to describe at least three different types of language used in graphic processing.

1. The input stream is in the form of actions taken by a console operator.
   a. draw with light pen
   b. type names and numbers
   c. push buttons
   d. light pen references of objects on the screen

   A language translator translates these actions into invocations of appropriate procedures. These procedures perform requested actions and provide displayed feed back to the user.
2. Input is in the form of pictures existing on film or other media. In this case the language translator is a pattern recognizer which recognizes and extracts meaning from these pictures.[5]
3. A set of programming tools (functions and subroutines) are embedded in a "host" language (e.g. FORTRAN). Using these tools lightens the load of the programmer of the graphic system.

The primary concern of this paper is the language and corresponding processor discussed above as type one. A set of service functions and subroutines (type three above) are also embodied in the system but are of secondary importance.

The system described here borrows freely from the best of earlier graphics work. It is set in an environment similar to GRAPHICS I[2] (i.e., a remote display with local compute power communicating with a central multiprogrammed computer). It uses a list structured data base, as expounded by Ling,[6] for storing both information about and relationships between graphic entities. This is similar to the "plex" structure used by D. Ross in the AED[3] system. This paper purports to contribute to the state-of-the-art by defining in Backus normal form[7] the class of language into which graphic statements may be imbedded, and then defining a language processing scheme that will translate any statement of this class. A useful feature of this scheme is that new graphic statements can be composed at the graphic display causing driving tables for the language processor to be produced automatically. Thus the language is automatically extendible within the class described below.

Graphic language

The class of language under scrutiny is a simple phrase structured grammar.[8] Statements of the language are of the following form:

Major Word <part>< >,

Minor Word <part>< > *Minor Word* etc.

The following are examples of some statements in the language:

Line <point definition> <point definition>
Joint <joint definition> ,
Radial Deflection <decimal number>

The language is defined in general terms as follows:

<Graphic Program> ::=
<Graphic Statement>|
<Graphic Program> <Graphic Statement>
<Graphic Statement> ::=
<Terminal Symbol> <Trailer>|
<Graphic Statement> <Terminal Symbol>
<Trailer>
<Trailer> ::= <Part>|<Trailer> <Part>|<Null>

The elements of the terminal vocabulary (denoted by <Terminal Symbol>) must be defined for the specific application. The portion of the non-terminal vocabulary that was left undefined <part> depends for definition upon the application and the specific graphic hardware. It consists of such things as <decimal numbers>, <light pen coordinate pairs> or <display item reference lists>.

Appendix A contains the Backus normal form language description of the graphic language designed for a large engineering design analysis system called "Multishell." Note that in this application members and joints are graphically just lines and points respectively. They may be defined as simply as:

1. Push "Position" button while pointing light pen at desired position on the screen
2. Push "Send" button

in which case the line is drawn from the last defined point to the new light pen position, or as laboriously as:

1. Point light pen at "Draw Member" menu item
2. Type x, y coordinates of start point
3. Type x, y coordinates of end point
4. Push send button

Note also that statement types (e.g. "Draw Member") are modal (i.e., do not have to be respecified before each subsequent statement).

Accepting the premise that any graphic statement will be an element of the class described above, we now attend to defining a language processing scheme that will translate these statements and compile useful code (display file) as a result.

Language processor

While it is, in principle, possible to let the input stream directly invoke the necessary procedures, it is convenient to interpose a lexical analyzer[9] between the graphic input devices and the procedures themselves. The duties of this lexical analyzer include isolating identifiers, literals, and the operators and delimiters of the graphic language. More important, however, it handles the switching of displayed menus as the user makes his selections with the light pen, and certain graphic manipulations (e.g., light pen tracking, picture moving, scaling, etc.). It also does some primitive syntax checking.

Since menu switching is one of the more significant duties performed by the lexical analyzer, it is embellished here. In appendix A, the "Case Identification Statement" contains the terminal symbol "Multishell" followed by a number of "parts." This statement is implemented as follows:

1. The following menu is displayed on the screen:
   SELECT A PROGRAM
   ADAM
   APT
   CYCLE
   DYNSAR
   MULTISHELL
2. The console user selects MULTISHELL by pointing the light pen at it.
3. This action by the user causes the lexical analyzer to delete that menu.
4. The terminal symbol *Multishell* is deposited in the attention file.
5. The following new menu is displayed:
   CASE IDENTIFICATION
   GO TO TYPEWRITER FOR
   FURTHER INSTRUCTIONS
6. The typewriter types
   IDENTIFICATION OF THIS CASE *
7. User types the identification code.
8. System user interaction continues until all "parts" of the case identification statement are supplied.

The language processor is modularized into two major subsystems. The first subsystem, *the lexical analyzer*, deals with the input language from the graphic console to a very shallow level. It is concerned only with the words or vocabulary of the language with little concern for the structure (syntax) and meaning (semantics) of the language. It recognizes inputs from the function keys, typewriter, and light pen, sorts them out and, where necessary, concatenates them into identifiers, real and integer literals, and the terminal symbols of the language.

The second subsystem, *the statement subroutines*, is concerned with the syntax and semantics of the graphic language. A statement subroutine exists for each statement in the language. It performs the actions indicated by the semantics of the statement. Many of the statement subroutines perform similar actions (i.e., manipulate or add to the data base, and add items

to the display file). Thus the library of subroutines may conveniently be divided into two classes (i.e., the statement subroutines themselves and the service routines and functions used in common by many of the subroutines). The statement subroutine may be thought of as the executive routine for that specific statement. In terms of language translation it performs syntactic analysis, code generation (display file), and the maintenance of a list structured data base. The service routines are a set of tools used by the statement subroutine in performing its functions. They provide a discipline towards generalization. As functions are found to be of common usage they are added to the service package. A block diagram of the system is illustrated by Figure 1.

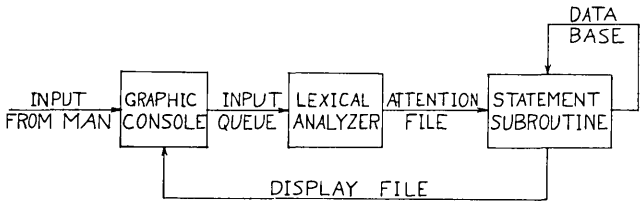


Figure 1 – System block diagram

The system is a simple feedback system where the man's input perturbs the system to a change in state which is reflected by a change in the display. The state of the machine is kept in the data base and the display file. The data base is list structured so that relationships that exist between displayed entities may be conveniently recorded.

The specific formats of all the data structures (i.e., the input queue attention file, data base, and display file) are machine dependent. However, their contents and logical structure are outlined here.

Input queue

The input queue receives its input directly from the graphic console. This is the input stream of characters to the entire system. There are several types of characters; function keys, alphanumeric keys, pen positions, and references to objects "seen" by the pen. The input stream is typically formatted one character per computer word with an identification field in each word to indicate which type of character it represents. In the interest of clarity and consistency an example will be introduced at this point and carried on through the description of all the data structures. The user "draws" a line on the console by the following actions, assuming the system has already been placed in the draw line *mode*:

1. typing 1.5, typing comma, typing 2.75,
2. positioning the pen at the line end point, and pushing the "position" button,
3. pushing the statement terminating "send" button.

This results in the input queue shown in Table I.

TABLE I – Input queue

Identification	Data
Alphanumeric	1
Alphanumeric	.
Alphanumeric	5
Alphanumeric	,
Alphanumeric	2
Alphanumeric	.
Alphanumeric	7
Alphanumeric	5
Function Key	Position
Pen Position	X Y
Function Key	Send

Attention file

The attention file is the output of the lexical analyzer. It is in a form that is convenient for the statement subroutines. The alphanumeric characters are concatenated into real numbers, integers, or symbols. The attention file, that results from lexical analysis of the input queue example above is illustrated by Table II.

TABLE II – Attention file

Type	Data
Terminal symbol	Line
Terminal symbol	X Y Coordinate pair
Real number	1.5
Real number	2.75
Terminal symbol	Position
Position	X Y
Terminal symbol	Statement Terminator

Data base

The data base is a list structured representation of the state of the graphic machine at any moment. The structure is hierarchical in that an item has either higher, lower, or the same level as any other item. Any number of relationships between items may exist.

The data base for the example statement is shown in Figure 2. The rectangular boxes represent items and the circular ones represent relationships and are called conjunctions.[6] The rings represent the list links.

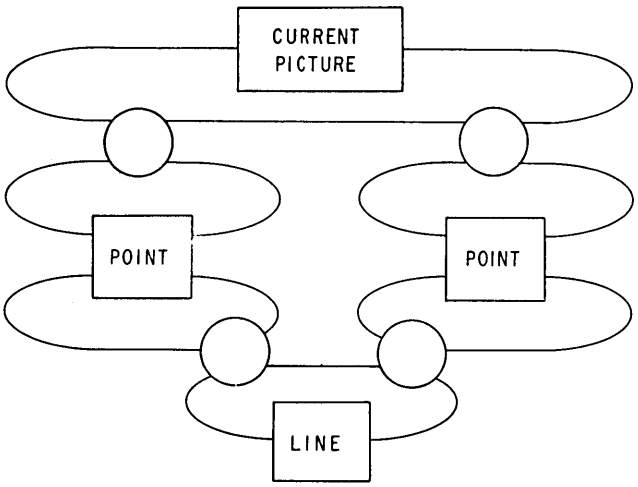


Figure 2 – Data base

Display file

The display file is the program which drives the graphic display. It contains such commands as draw line, draw point, or print character. The display file that results from our example is shown in Table III.

TABLE III – Display file

Identifier	Data	
Position Beam to Draw Point	X	Y
Draw Line to Draw Point	X	Y

This is the final display file entry that results from our example. The lexical analyzer, while processing our input, forms some temporary display file entries e.g., the alphanumeric characters, points, etc., but these are subsequently removed when the statement subroutine completes its job.

Lexical analyzer

The lexical analyzer has seven major duties.

1. Concatenates literals (real, integer) and symbols.
2. Recognizes terminal symbols. Identifies them as such so that the proper statement subroutine may subsequently be called.
3. Recognizes statement termination and as a result causes the appropriate statement subroutine to be called.
4. Maintains a temporary display file for editing purposes (points, characters, operator language guides).
5. Produces as output the attention file (AF) to be passed along to the statement subroutines.
6. Displays new menus as selections are made.
7. Does some primitive syntax checking.

The fact that the lexical analyzer is required to know very little about the meaning of the language it processes, allows it to be written very generally. It can, in fact, be practically language independent. This means that adding new language requires no change in the lexical analyzer itself. Instead appropriate additions are made to its driving tables. Figure 3 shows the lexical analyzer, its associated tables, and how it fits in with the rest of the system.

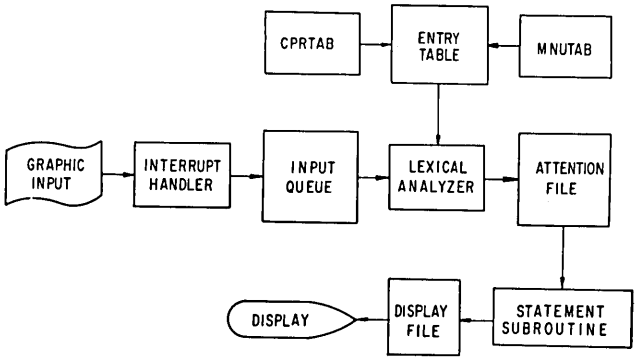


Figure 3 – Lexical analyzer

The lexical analyzers operation is described as follows. The lexical analyzer operates between two ring buffers. It gets its input stream of characters from the *Input Queue*. It is driven by the *Entry Table* which contains the entries that the lexical analyzer must make in its output files as a result of each possible input. The specific entry in the Entry Table is pointed to by either the *Character Property Table* for function keys or alphanumeric keys or by the *Menu Table* for menu references. The output files that are driven by the lexical analyzer are the Attention File and the Display File. The *Attention File* is the other ring buffer mentioned above and passes the lexically analyzed information on to the Statement Subroutines. Some statements also cause output to be put into the *Display File* (i.e., changes in display of menus or immediate reactions to user actions).

The contents and logical structure of the *Entry Table* are illustrated below by Table IV.

TABLE IV – Entry table

No. of Phrases in this statement Increment added if this is a phrase No. of parts in this phrase	Header
No. of words for Attention File No. of words for Display File	
Enable teletype flag Enable light pen flag	
Attention File Entries	Header
Display File Entries	

The Entry Table is the lexical analyzers key driving table. The first three words allow the lexical analyzer to do its primitive syntax check. This consists of checking to see that the statement has the correct number of phrases and that each phrase has the correct number of parts. The entries are moved by the lexical analyzer to the appropriate output file.

The logic of the Lexical Analyzer is illustrated by Figure 4.

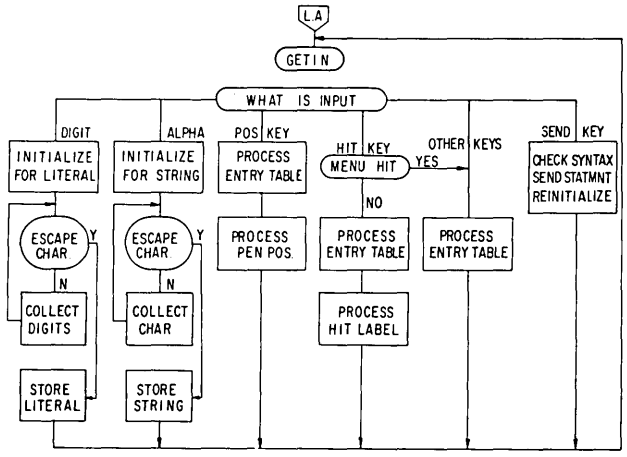


Figure 4 – Lexical analyzer logic

Statement subroutines

The statement subroutines consist of the application routines and a package of general purpose subroutines. A detailed description of all statements in a specific application is outside the scope of this paper. However, a flow diagram of the skeleton of a statement subroutine is illustrated by Figure 5.

The general purpose service subroutines fall into two classes:

1. The list processing routines are used for creating, manipulating (e.g., searching, relating, etc.) and destroying lists.
2. The display file processing routines are used to display items and remove items from the display.

Environment

The proposed environment of the system is described as follows. A small digital computer is placed at the end of the remote phone line with the graphic console. It, as well as other remote terminals (e.g., teletypes and high speed printer, card reader combinations) communicate with the GE 635 central computer through voice grade lines at 2.0 kb rate. The GE 635, operating under the GE Comprehensive Operating Supervisor and the GE Remote Terminal Supervisor, provides a multiprogrammed remote terminal environment for the system. The interface between the main frame and the remote processor is now at the attention file and the display file. That is the input queue, lexical analyzer, attention file and display file are located in the remote processor. The statement subroutines, data base, and attention file are located in the main frame. With this division of responsibility, the remote processor does the tasks that require only a shallow understanding of the problem i.e., the lexical analysis. The main frame contains the data base and statement subroutines and does the more detailed processing. This system is pictured in Figure 6.

Application flexibility

One important practical consideration that has not yet been mentioned is the question of application flexibility. The lexical analyzer embodies a flexible table driven translation methodology. However, coding these tables by hand for every menu of every application would be a tremendously tedious task. Therefore, a convenient means of building these tables needs to be provided.

Ideally, the application programmer would like to sit at the graphic display and compose his menus. But, provision must also be made for labeling items in a menu, cross referencing them to other menus, and indicating the function code by which the statement subroutine will recognize the item in the attention file. A sub language is proposed which uses the lexical analyzer itself to generate new menus. Of course, the tables for this sub language must be coded by hand. But, once the system is thus bootstrapped, new applications can be added easily.

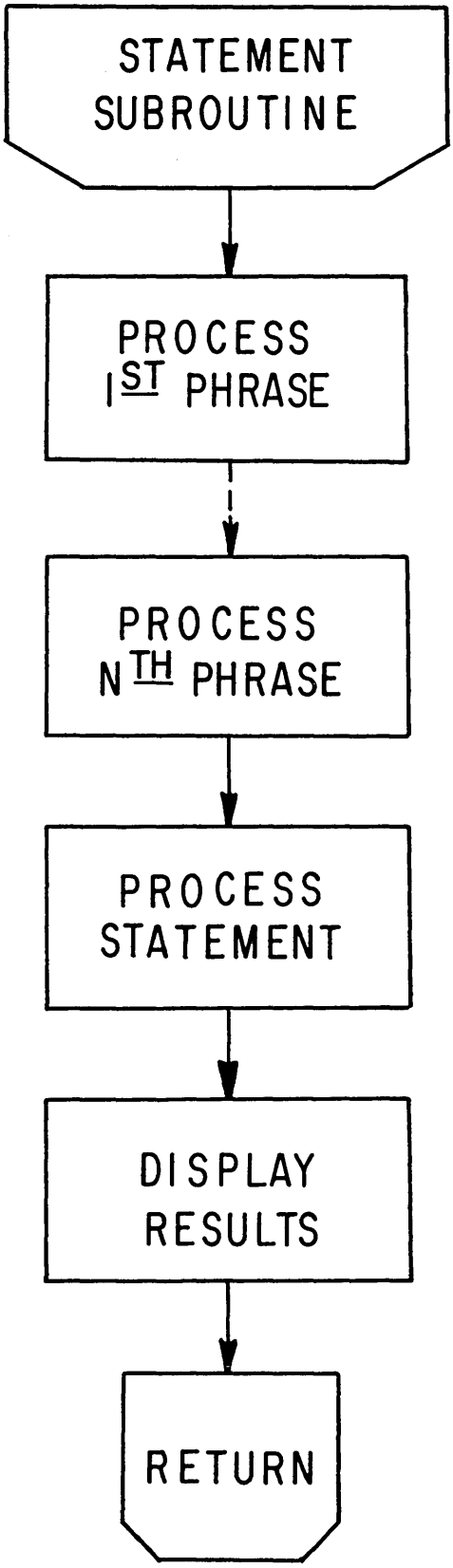


Figure 5—Statement subroutines

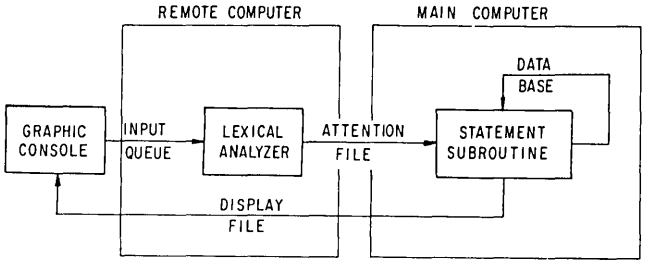


Figure 6—System Environment

SUMMARY

The major premise on which this paper is based is that all graphic statements fall into the class described earlier. If the potential user of the system will accept this restriction, his application can be implemented with a minimum of effort. He must write the statement subroutines, which he would have to do in any case. Then he must compose the language by which he will communicate with these subroutines. He does this graphically at the graphic console. The software system does not change from application to application except for the statement subroutines and the contents of the lexical analyzer's driving tables.

Both D. Ross with AED[3] and L. Roberts with extensions to VITAL[10] have taken approaches similar to that described in this paper. In both cases, however, graphic language translation ability was added to existing syntax directed compilers. This paper presents a translating scheme that is specifically addressed to translating a broad class of user oriented languages in a multi-computer graphic network. It is dedicated to the proposition that many specific application oriented languages are superior to one generalized language. It is based on the premise that the only things common among these many application oriented language systems are the lexical analyzer and a set of service routines. So it proposes a lexical analyzer in a remote processor and a set of service routines in the main computer for the application programmer to use to implement his graphic interaction. action.

ACKNOWLEDGMENTS

The author gratefully acknowledges the many people whose ideas have contributed to this paper. Chief among these contributors were A. Dean, M. Ling, and G. Link.

REFERENCES

1 I E SUTHERLAND
Sketchpad, a man-machine graphics communication system
SJCC vol 23 Spartan Books Inc Washington D C 1963

2 W H NINKE
Graphics I—a remote graphical display console system
Proceedings of the FJCC Las Vegas Nevada December 1965

3 D T ROSS
AED user kit
Massachusetts Institute of Technology, Electronic Systems Laboratory, various memoranda

4 E L JACKS
A laboratory for the study of graphical man-machine communication
Proceedings of the FJCC San Francisco California October 27-29 1964

5 R NARASIMHAN
Syntax-directed interpretation of classes of pictures
Comm. ACM, vol 9 no 3, March 1966

6 T S LING
General Electric reactive display system
Proceedings IEEE Region III Convention, Atlanta, Georgia, 1966

7 J W BACKUS
The syntax and semantics of the proposed international language of the Zurich ACM-GAMM conference ICIP
Paris, France. June 1959

8 CHOMSKY
On certain formal properties of grammars
Information & Control no 2 vol 2, 1959

9 T E CHEATHAM, JR.
Notes on compiling techniques
University of Michigan, Summer Conference Course on Automatic Programming, June 1966

10 L G ROBERTS
A graphical service system with variable syntax
Comm. ACM, vol 9 no 3, March 1966

Appendix A

Multishell statement definitions

General

<Multishell Program> :: = <Case Id. St.> *Send* <Structure Definition>
<Structure Definition> :: = <Structure Drawing> <Property Definition>
<Property Definition> :: = <Property Def. St.> | <Property Definition> <Property Def. St.>
<Property Def. St.> :: = <Boundary Cond. St.> *Send* | <Member Prop. St.> *Send*
<Structure Drawing> :: = <Structure St.> *Send* | <Structure Drawing> <Structure St.> *Send*

Case identification

<Case Id. St.> :: = *Multishell* <Id.><Name><Ext.><Mail Drop><Date><Ref. RPM><Ref.Temp.>
<Id.> :: = <Long String>
<Name> :: = <Long String>
<Ext.> :: = <Short String>
<Mail Drop> :: = <Short String>
<Date> :: = <Long String>
<Ref. RPM> :: = <Number>
<Ref. Temp.> :: = <Number>

Structure drawing

<Structure St.> :: = *Draw Member* <Point Part> | <Structure St.> <Point Part>
No more than two point parts
<Point Part> :: = <Point Def.> | <Point Ref.>
<Point Def.> :: = <Light Pen Coord. Pair> | <Typed Coord. Pair>
<Erase St.> :: = *Erase* <Ref.>
<Scale St.> :: = *Scale* <X−small> <X−large> <Y−small> <Y−large>
<X−small> :: = <Number>
<X−large> :: = <Number>
<Y−small> :: = <Number>
<Y−large> :: = <Number>

Boundary conditions

<Boundary Cond. St.> :: = *Joint* <Point Ref.> <Boundary Conditions>
<Boundary Conditions> :: = <Radial Part> <Axial Part> <Rotational Part>
<Radial Part> :: = <Radial Type> <Number> | <Null>
<Axial Part> :: = <Axial Type> <Number> | <Null>
<Rotational Part> :: = <Rotational Type> <Number> | <Null>
<Radial Type> :: =  |  |  | 
<Axial Type> :: =  |  |  | 
<Rotational Type> :: = | | |

Member properties

<Member Prop. St.> :: = *Member* <Line Ref.> <Properties>
<Properties> :: = <Material Prop.> <Thickness> <Rigidity> <Environment>
<Material Prop.> :: = *M Data* <Number> | *Material Properties* <Poisson's Ratio>
<Mass Density> <I end E> <I end A> <J end E> <J end A>
<Poisson's Ratio> :: = <Number>
<Mass Density> :: = <Number>
<I end E> :: = <Number>
<I end A> :: = <Number>
<J end E> :: = <Number>
<J end A> :: = <Number>

<Thickness> :: = *Thickness* <I end Thickness> <J end Thickness>
<I end Thickness> :: = <Number>
<J end Thickness> :: = <Number>
<Rigidity> :: = *Rigidity* <I end constraints> <J end constraints>
<I end constraints> :: = ↑ | → |) | Null
<J end constraints> :: = ↑ | → |) | Null
<Environment> :: = *Environment* <Pressure> <Temp. I end> <Temp. J end>
<Thermal Gradient>
<Pressure> :: = <Number> | <Null>
<Temp. I end> :: = <Number> | <Null>
<Temp. J end> :: = <Number> | <Null>
<Thermal Gradient> :: = <Number> | <Null>

Graphic language primative definitions

Numbers

<Number> :: = <Unsigned decimal number> | <Adding Operator> <Unsigned decimal number>
<Unsigned decimal number> :: = <Decimal Number> <Exponent Part>
<Decimal Number> :: = <Unsigned Integer> .|. <Unsigned Integer> | <Unsigned Integer> .
<Unsigned Integer> | <Unsigned Integer>
<Exponent Part> :: = <Null> | E <Integer>
<Integer> :: = <Unsigned Integer> | <Adding Operator> <Unsigned Integer>
<Unsigned Integer> :: = <Digit> | <Unsigned Integer> <Digit>
Limit of 6 digits
<Digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Adding Operator> :: = + | −

Names

<Short String> :: = '<Basic String>'
Limit of 6 Char.
<Long String> :: = '<Basic String>'
Limit of 24 Char.
<Basic String> :: = <Char.> | <Basic String> <Char.>
<Char.> :: = <Digit> | <Letter> | <Special Character>
<Letter> :: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<Special Character> :: = SP|!|"|#|$|%|&|(|)|*|+|,|−|.|/|:|;|<|=|>|?|@|[|\|]|↑|←

Graphic primatives

<Typed Coord. Pair> :: = <Number> <Number>
<Light Pen Coord. Pair> :: = *Light Pen Position* *Position*
<Ref.> :: = <Point Ref.> | <Line Ref.> | <Item Ref.>
<Point Ref.> :: = *Light Pen Point* *Hit*
<Line Ref.> :: = *Light Pen Line* *Hit*
<Item Ref.> :: = *Light Pen Item* *Hit*
<Null> :: =

Design of fault-tolerant computers

by ALGIRDAS AVIŽIENIS
University of California
Los Angeles, California, and
Jet Propulsion Laboratory
Pasadena, California

Causes and symptoms of logic faults in digital systems

Reliable performance of hardware has been a requirement for digital systems since the construction of the first digital computer. Improper functioning of the logic circuits in a digital system is manifested by *logic faults,* which are defined for this paper as "permanent or transient deviations of logic variables from the values specified in design."

Permanent faults are caused by physical changes in the components of a logic circuit which permanently alter the logic function specified by the designer. The most common permanent faults are the *determinate* faults of "stuck on zero" and "stuck on one" types. Less frequent is the *indeterminate* or "stuck on X" fault, in which the logic variable assumes both "zero" and "one" values improperly during a sequence of operations. Faults also differ in their extent: a *local* fault affects only one logic circuit, while a *distributed* (or catastrophic) fault occurs when one failure creates faults in several logic circuits of the same system.

Transient faults are caused by temporary changes in the properties of logic circuits, which lead to deviations from the specified values of logic variables. Such transient faults also belong to one of the above listed categories. They are caused either by external influences (electromagnetic interference, noise in power supply, etc.) or by temporary circuit malfunctions (overheating, overload conditions, etc.). External causes may simultaneously induce many faults throughout the system, therefore independent occurrence cannot be assumed for all transient faults.

At the system level a logic fault is manifested as an *error* in the program being executed by the system. Two types of errors may be distinguished. A *word error* occurs when a computer word (data word or instruction) is altered by a fault. A description of the alteration is called the *damage pattern* of the fault. A *logic error* occurs when an individual logic variable, which is not a part of a structured word, is altered by a fault. Examples of such variables are the various control signals in a computer. A logic error alters the algorithm being executed in some undersirable manner.

The present paper is concerned with the introduction of fault-tolerance in order to increase the reliability and availability of a digital system. We say that a system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults. All above discussed types of faults and errors need to be considered in the design of a fault-tolerant computer. Theoretical studies of fault-tolerance need a clear identification of the types of faults and/or errors which are to be tolerated.

Protective redundancy for fault-tolerance

Reliable performance of digital systems is usually attained by the systematic application of two techniques. The first is the selection of highly reliable components and the use of proven methods for their interconnection and packaging. The second technique is an extensive verification of the logic design, of the programs, and of the finished hardware, first by simulation and later by diagnostic and functional tests under expected environmental conditions. In spite of these reliability assurance techniques, the system may still fail during use because of uncontrollable or undetected faults. These are caused by undetected design errors, random failures of components or connections, and externally induced malfunctions during the operation of the system.

The effects of logic faults can be eliminated by the introduction of *protective redundancy* into the system. A computer system contains protective redundancy if faults can be tolerated because of the use of additional components or programs, or the use of more time for the computational tasks. These additional components, programs, and time are not required by the system in order to execute the specified tasks as long as faults

do not occur. The techniques of protective redundancy may be divided into two major categories: *massive* (also called *masking*) redundancy and *selective* redundancy.

In the massive (masking) redundancy approach the effect of a faulty component, circuit, signal, subsystem, or system is masked instantaneously by permanently connected and concurrently operating replicas of the faulty element. The level at which replication occurs ranges from individual circuit components to entire self-contained systems. Theoretical studies of massive redundancy were initiated in January 1952 by John von Neumann in a series of five lectures at the California Institute of Technology.[1] Other pioneering contributions in this field are due to C. J. Creveling,[2] and to E. F. Moore and C. E. Shannon.[3] The subject has attracted considerable attention in the past decade, and four principal techniques of massive redundancy may be distinguished among the published studies. These techniques are:

1. Replication of circuit components: e.g., "quadded" diodes, resistors, transistors, duplicated connections, etc.[2,4]
2. Replication or coding of logic signals: use of multiple channels and voting elements,[1,5,6] recursive nets,[3,7] interwoven logic,[8,9] variation-tolerant coded threshold element nets.[10]
3. Adaptive logic elements, e.g., voters with variable-weight inputs.[9]
4. Replication of entire systems with comparison or voting at system level.[11,12,13]

The category of selective redundancy encompasses redundancy techniques which fall outside the definition of massive redundancy. Since instantaneous masking is excluded in the selective technique, it is necessary to detect the presence of a fault. Subsequently, the fault is made harmless by a corrective action. The techniques of fault detection fall into two major categories:

1. *Concurrent diagnosis* by the application of error-detecting codes and special monitoring circuits. Detection occurs while the system is being used.[14,15,16,17,18]
2. *Periodic diagnosis* using diagnostic hardware and/or programs. Use of the system is interrupted for diagnosis.[19,20,21,22,23]

A variety of approaches exists in the implementation of both methods; furthermore, combinations of both techniques have been successfully employed.[24]

A *corrective action* which eliminates effects of the fault must follow the detection. The four principal techniques of correction are:

1. Correction of errors by the use of error-correcting codes and associated special purpose hardware and/or software (including recomputation).[15,18,24]
2. Replacement of the faulty element or system by a stand-by spare.[25,26,27]
3. Replacement as above, with subsequent maintenance of the replaced part and its return to the stand-by state.[28,29,30]
4. Reorganization of the system into a different fault-free configuration which can continue the specified task.[31,32,33]

It must be emphasized that the preceding classification of protective redundancy techniques is intended to facilitate a systematic approach to the study of fault-tolerant systems. In most practical cases a mixture of these techniques has been proposed or used in order to attain fault-tolerance. The references which are cited in this section contain fundamental contributions to the theory of protective redundancy, and have been chosen to serve as illustrations of the various approaches. However, this is not an exhaustive listing of all relevant contributions, and the reader is referred to bibliographies in the cited references for further papers. Especially the book by W. H. Pierce[9] and the recent study by J. Goldberg, et al.,[33] contain very extensive bibliographies on protective redundancy. The latter study includes numerous references to Russian publications in this field as well as reviews of various techniques.

State of the art in the design of fault-tolerant computers

The continuing increases in the speed and complexity of digital systems accelerate the demand for fault-tolerance. The cost of uncorrected errors is especially severe in large time-shared computer service systems and in situations in which a computer controls a very valuable system, and is not accessible to human repair. Examples are a real-time control computer and a spacecraft computer controlling an interplanetary mission. A second critical requirement for fault-tolerance exists when human lives may be affected by computer errors, e.g., in military defense systems and in control of high-speed transportation or of medical systems.

The most immediate solution in such critical applications has been the replication of entire systems,[11,12,13] frequently backed up by transfer of control to a human operator or to a separate, less precise backup system. The replication at system level becomes extremely costly when very large and fast systems(e.g., time-shared "computer utility" systems) must be replicated. Furthermore, occurrence of independent faults in two or more replicas is more probable as the systems become more complex or as the required unattended lifetime is increased. The need for lower cost of protective redundancy and for longer mean life values of protected

systems has stimulated studies of other methods of fault-tolerance.

In the early stages of development attention had been directed toward massive redundancy at the lowest level —the replication of individual components (resistors, transistors, etc.).[2,3,4] The principal example of its practical application is the primary processor of the Orbiting Astronomical Observatory.[4] The use of component redundancy has been limited by design difficulties and by new developments in component technology. The design of logic circuits becomes very difficult because the circuits must function correctly under wide variations of component values, caused by shorting or opening of individual components. The change from discrete components to integrated circuits has largely invalidated the assumption of independent component failures. Without it, the advantages of component redundancy are lost.

Considerable effort has been continuously directed toward practical use of massive triple modular redundancy (TMR) in which logic signals are handled in three identical channels and faults are masked by vote-taking elements distributed throughout the system.[5,6,9] Studies have considered optimization of voter placement and analyzed the gain in reliability or mean life of a TMR system.[34,35,36,37] In the most important application to this date, the guidance computer for the Saturn V spacecraft launch vehicle has been designed employing TMR techniques in its arithmetic and control sections.[6,38] This large-scale application of TMR may be expected to provide a practical assessment of its effectiveness.

TMR and related types of massive redundancy at the level of logic signals offer both obvious advantages and some serious drawbacks in a general comparison to selective redundancy. The principal advantages of massive redundancy are:

1. The corrective action is immediate and "wired-in," while it is delayed and may require switching in selective redundancy.
2. During operation there is no need for fault detection, which is essential in selective redundancy.
3. All parts of the system are equally protected; unprotected "hard core" elements may exist only at interfaces with other systems. In most selective redundancy schemes a "hard core" exists in the system.
4. The conversion of a non-redundant design to a massively redundant one is relatively straightforward, while more novel design techniques are demanded by the introduction of selective redundancy.

Compared to massive redundancy, the selective form requires several additional features: a system ability to tolerate interruptions for repair and to execute a "roll-back" for error correction, sophisticated diagnosis methods, protection for the "hard core," and trade-off studies between time, program, and hardware replication. The advantages of selective redundancy over the massive form are, however, also very significant in most applications:

1. Power is required by only one copy of each replaceable item in a replacement system; all parts require power in the massive form.
2. The replacement switch provides fault isolation between subsystems; such isolation is essential in the case of catastrophic failures. Massive redundancy usually assumes independent failures of logic elements; such independence requires isolation which is difficult to provide for batch-fabricated integrated circuit packages. The entire batch may possess the same defect; also, mechanical or thermal damage is likely to affect an entire package, rather than single logic circuits.
3. All spares can be utilized in selective redundancy; in the massive form a majority of faulty elements in a given region leads to system failure.
4. The designs of individual replaceable blocks may be altered, and the number of spares may be adjusted to a given requirement without changes in the system design in the case of selective redundancy; such changes are more difficult in the massive case.
5. The replication in massive redundancy frequently leads to increased fan-out and fan-in requirements for logic elements, or to increased tolerance limits in circuit design; such problems are avoided in the selective case.
6. Permanent connection of the redundant elements makes the initial check-out more difficult to implement in systems with massive redundancy; special circuits and system outputs are necessary.
7. Massively redundant systems with voting require synchronization of the separate channels at the voting elements; they also are susceptible to transient external influences (e.g., sparks) which alter logic signals in a majority of channels without leaving permanent damage. The delayed occurrence of diagnosis in the selective case allows detection of such transient changes in signals.

The most developed techniques of selective redundancy are fault detection by periodic diagnosis and the application of parity and similar error codes[6,14] to detect or correct errors in data transmission and storage. The periodic diagnosis techniques have progressed from exclusively software implementations to software combinations with special-purpose hardware[21,24] and to studies of system design methods which facilitate their self-diagnosis.[22,23] An extensive bibliography on periodic

diagnosis has been compiled by Breuer.[39] Except for parity checking, there have been relatively few studies of concurrent diagnosis in an entire computer including the processors,[17] or in a generalized logic net.[15,18]

Theoretical results on general models of replacement systems demonstrate gains in reliability and mean life for both unmaintained [25,26] and maintained[28,29,30] cases. More recently there have been proposals for designs of computers as replacement systems.[40,41] The following sections of this paper present a replacement system with concurrent diagnosis[27] which is presently being constructed. Reorganization of a system upon fault detection has also been discussed in recent publications;[31,32,33] however, much work remains to be done in order to arrive at complete system designs and to derive measures of expected effectiveness.[31,42]

In conclusion it is noted that the rapid development of integrated circuit technology is causing a shift of emphasis from massive to selective redundancy techniques in which functional units of a system are replaceable as single elements. It is also possible that massive redundancy will find a new application in large-scale integration, serving to mask manufacturing defects and thus increasing the yield of the manufacturing process.

Design considerations for a fault-tolerant spacecraft computer

Theoretical studies of selective redundancy, and specifically of replacement systems, indicate that a significant increase in the availability and in the mean life of a digital system may be attained without the high cost of complete replication and concurrent operation of several copies of a system. The challenge to the designer at the present time is to create computer systems which translate the theory into a working system which uses state-of-the-art components, meets current performance requirements, and attains the theoretically possible gains in reliability.

The choice of a method or of a combination of methods of redundancy for a particular computing system is influenced by the intended application. The present section considers the application of protective redundancy to a guidance and control computer for an unmanned interplanetary spacecraft. The computer may also be employed for the onboard processing of scientific data when guidance computation is no in progress. The guidance computer is required to survive space voyages to other planets which range up to several years in duration and to perform approach guidance and control computations at the end of the voyage. Continued control of the spacecraft after arrival, processing of scientific data collected, control of the landing and operation of a capsule, and guidance for a return voyage may also be required. Course corrections are to be computed one or more times during the mission; considerable time is available for this task. The computing at launch and in early stages of the mission may be performed or supported by computers on the ground and in the launch vehicle. The very long communication distances and possible occultation of the spacecraft make Earth-based support ineffective at the destination planet. The computations which are required at a remote destination present the most demanding problem to the spacecraft guidance and control computer.

The design of a spacecraft computer must be performed within the strict constraints of the available power, weight, and volume. The existence of these constraints indicates an advantage for selective redundancy, which does not necessarily require power for the spare replicas and which offers protection with the minimum of one spare for each operating element. An evaluation of relative advantages of the massive and selective redundancy approaches has led to the choice of selective redundancy for fault-tolerance in a spacecraft guidance computer which is being developed at the Jet Propulsion Laboratory. It will be called the "JPL Self-Testing And -Repairing" (abbreviated JPL-STAR) computer in this paper. The performance requirements demand a certain computing capacity at the end of a long voyage, and there are no requirements for a higher computing capacity at an earlier time during the mission. Under these conditions, a fixed-configuration replacement system possessing the required capacity is preferred over a reorganizable or "degradable" system which has a minimal configuration of the same capacity. The replacement system is a simpler solution, since it avoids the programs, switches, and control hardware which perform the reconfiguration and resulting rescheduling of operation.

A replacement system provides to the user one standard configuration of functional subsystems which has the required computing capacity. The standard computer is supplemented by one or more spares of each subsystem. The spares are held in a standby condition and serve as replacements of operating units when permanent faults are discovered. The presence of spares imposes additional requirements on the selectively redundant system. In addition to the ordinary functions of a computer, the system must incorporate some means of fault detection, a recovery procedure for the case of transient faults, a replacement procedure and a switch for the case of permanent faults, and a checkout procedure for all spares before the mission. The standard configuration itself must be designed as an array of self-contained functional subsystems with clearly defined interfaces for replacement switching. The hardware or software which controls the recovery and/or replacement must be fault-tolerant as well.

Early in the design fundamental choices must be made between hardware and software implementations of the fault detection and recovery procedures. The current generation of aerospace computers almost exclusively uses software techniques, supplemented by hardware for parity checking of data storage and transfer. The continuing decrease in the size and in power requirements of integrated electronic circuits, as well as the vulnerability of software techniques in the case of memory failures, led to the choice of a hardware implementation of fault detection and recovery in the JPL-STAR computer. The experimental breadboard Model I JPL-STAR system is expected to provide valuable operational experience about such an extensive use of hardware techniques in a replacement system. An actual hardware design rather than simulation was chosen in order to explore the circuit aspects of switching, fault detection, isolation of faulty subsystems, and recovery from transient faults.

Fault detection in digital circuits is implemented either by periodic or by concurrent diagnosis. The currently most common approach is periodic diagnosis which utilizes a diagnostic program stored in the memory. Computation is periodically interrupted and the diagnostic program is executed. Detection of a fault initiates the replacement procedure; the program is "rolled back" to a point preceding the previous (successful) diagnosis period. Errors in computation which have been caused by transient faults remain undetected in this approach. The diagnosis program itself is vulnerable to faults in the memory system. The cost of diagnosis consists of the storage used for the diagnostic program, of the time consumed by its execution and of the time needed for repair and repeated execution of the program segment which was run after the last diagnosis. Such time costs are very severe in re-entry and landing programs for guidance and control which require real-time computing. The alternate diagnosis method is concurrent diagnosis in which error-detecting codes and monitoring circuits are employed to show the presence of faults. The execution of every instruction is checked immediately; instead of the stored diagnostic program, the cost is in hardware and consists of the logic circuits which perform the code checking algorithm and the other monitoring circuits. Errors due to transient faults are detectable, and the immediate detection of a fault permits a relatively short rollback of the program. For these reasons concurrent diagnosis has been chosen for fault detection in the JPL-STAR computer. The simplest and most costly error-detecting code (100% redundancy) is the complete duplication of program and data words. Errors are indicated by the disagreement of two words; further diagnosis is needed to pinpoint the faulty source. Parity and other more complex codes which detect errors in the transmission of digital data have a lower redundancy, but are not suitable for the checking of arithmetic operations. In order to apply a uniform code in the entire system, arithmetical error-detecting codes were selected as a means of concurrent diagnosis for the JPL-STAR system. An extensive theoretical investigation of the effectiveness, cost and applicability of arithmetic codes was conducted prior to the system design of the JPL-STAR computer.[16,17] The results showed the existence of a class of low-cost codes with sufficient effectiveness of error detection for concurrent diagnosis. The code-checking circuits are supplemented by monitoring circuits which verify the synchronization of operation for the various subsystems. Other circuits compare duplicated critical functions of the subsystems and measure important circuit parameters (e.g., read and write currents in memory units). The monitoring circuits are included in order to detect the faults which are not always indicated by the code checking algorithm.

Recovery and replacement procedures require both software and hardware contributions. Consistently with the choice of hardware for fault detection, the JPL-STAR computer employs hardware implementation to the furthest possible degree in these procedures as well. The most fundamental hardware consideration in a replacement system is the method of switching and the nature of the switch which implements the replacement operation. The reliability of the switch is a limiting factor in the estimates of reliability for the entire system. Furthermore, the switch must provide complete isolation in the case of catastrophic failures occurring in the part of a computer which is to be replaced. The principal alternatives in the choice of a switching method are *information switching* and *power switching*. A study of switching techniques[43] has led to the conclusion that the switching of power to replaceable units offers strong isolation against catastrophic failures and minimizes the number of switches requiring extreme reliability. Furthermore, the data transmission speed within the computer is not affected by the circuit properties of the switch. A magnetic power switch for the JPL-STAR computer which is an integral part of a replaceable unit has been designed and constructed. The switch is a part of the unit's power supply and is designed to fail asymmetrically—in an open mode.

The use of a power switch requires that all unpowered copies of a replaceable unit should be permanently attached to the data transmission busses of the system. As a consequence, an unpowered unit is required to produce only logic signals of value "zero" on all of its output lines. Furthermore, all input and output lines of every replaceable unit are isolated from the busses

in order to prevent shorting of a bus by a short inside the unit.

Organization of the JPL-STAR computer, model I

The preceding section has outlined the principal alternatives which were considered in the choice of fault-tolerance techniques for the JPL-STAR computer. An experimental breadboard Model I of the JPL-STAR computer has been designed and is presently being constructed. The main objectives of the Mod I STAR computer are to gain experience with the hardware aspects of a replacement system and to conduct experiments with fault detection and recovery procedures. The performance specifications of the Mod I STAR computer are similar to those of many present-generation aerospace computers; they have not been matched to any specific application. The fundamental choices in fault-tolerance techniques are as follows:

1. All machine words (data and instructions) are encoded in an error-detecting code.
2. The computer is subdivided into several replaceable functional units.
3. Fault detection, recovery, and replacement are carried out by special-purpose hardware; software techniques may be added later to provide additional fault-tolerance features.
4. Replacement is implemented by power switching: units are removed by turning power off, and connected by turning power on.
5. The information lines of all units are permanently connected to the busses through isolating circuits; unpowered units produce only logic "zero" outputs.
6. The error-detecting code is supplemented by monitoring circuits which serve to verify the proper synchronization and internal operation of the functional units.

The Model I employs a 32-bit word length for its operands and instructions. Machine words are transmitted between the functional units in four-bit bytes, that is, in a series-parallel mode. The functional units contain their own sequence generators and possess identical input and output connections. A typical functional unit is shown in Figure 1. The "Info. Input" and "Info. Output" lines are connected to information busses. They receive and send coded machine words, one byte at a time. The "Switch Control" line supplies the "change position" command to the power switch, while the present switch position is shown by the "Switch Status" line. The other control input lines supply a "Clock" pulse train input, a synchronization test signal ("Sync"), and a "Reset" signal which places the functional unit into a standard state. There are also three more status output lines. The "Active" signal

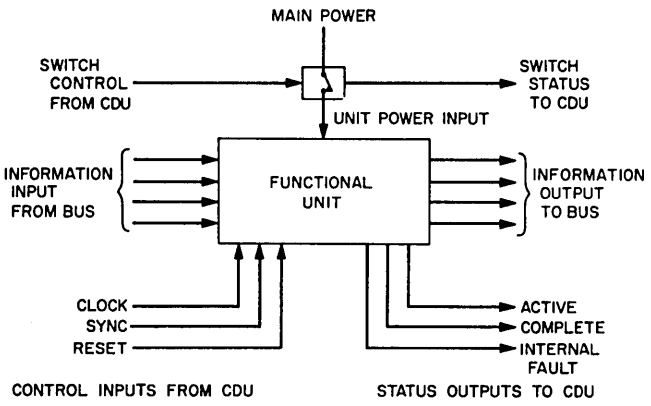


Figure 1—Typical JPL-STAR computer functional unit

indicates that at least one Info. Output line has an active (logic "one") output. The "Complete" signal occurs at the end of every subalgorithm being performed by the unit. The "Internal Fault" signal occurs when the internal monitoring circuits of the unit detect an abnormal condition. All status outputs are connected to the Control and Diagnosis Unit (CDU). The CDU also generates the four control input signals. The CDU initiates all recovery and replacement actions on the basis of the status signals reecived from the powered functional units in the system.

The block diagram of the JPL-STAR Model I computer is shown in Figure 2. It is a fixed-point, binary computer suitable for spacecraft guidance applications. Information words are transmitted on two busses in bytes of four bits each. The choice of the byte mode reduces the size of busses and simplifies the checkers, which are diagnostic hardware for error detection in the transmitted information words. An expansion to parallel operation is straightforward and will increase the computing speed at the cost of larger busses and more complex checkers. The replaceable functional units of Model I are:

1. a main arithmetic processor (MAP);
2. a control arithmetic processor (CAP);
3. a 16K read-only memory unit (ROM);
4. up to 12 read-write memory units, 4K each (RWM);
5. an input/output (buffer) unit (IOU);
6. a logic processor (LOP);
7. an interrupt unit (IRU);
8. a system clock unit (SCU);
9. two bus checkers (CH1, CH2);
10. a control and diagnosis unit (CDU).

Properties of these functional units are summarized in the next section.

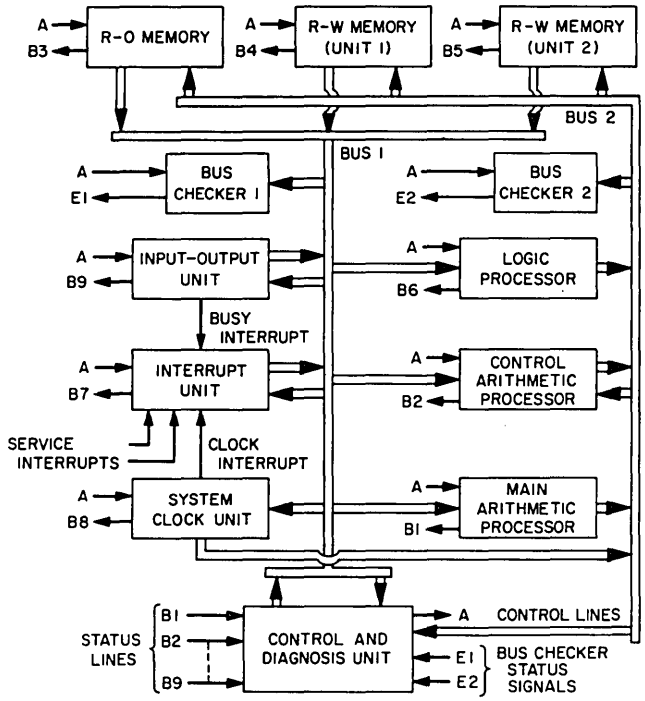


Figure 2—JPL-STAR model I computer block diagram

All information words in the JPL-STAR computer are encoded in an error-detecting code. In the case of numerical data words and addresses of instructions the code must be preserved during arithmetic operations. The two principal methods of arithmetic encoding are product (or "An") and residue codes.[16,17,44,45] In order to gain a better understanding of the relative virtues of these two methods, both are employed in the Model I: product coding for numeric operands, and residue coding for addresses. Figure 3 shows the formats of numeric operands and instructions.

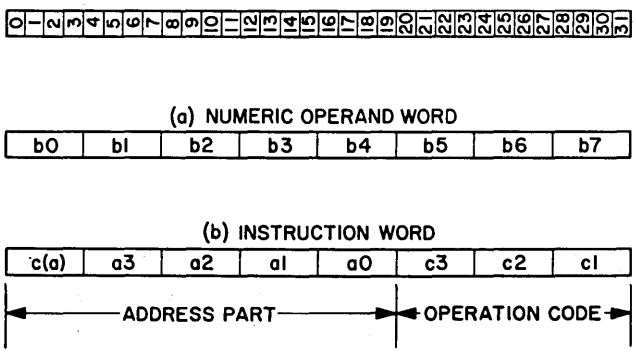


Figure 3—JPL-STAR model I computer word formats

The numeric operands (Figure 3a) are 32 bits long binary product-coded numbers with the check factor 15. Binary numeric operands x (28 bits long) are multiplied by 15 to obtain the product coded 32-bit operands 15x. The check factor 15 has been found to be especially effective in the case of series-parallel transmission and computing in bytes of 4 bits length.[16] The *checking algorithm* computes the modulo 15 residue of coded words which are transmitted on the busses. A zero residue (represented by 1111) indicates a coded word; all other values indicate a fault in the functional unit which delivered the word to the bus.

The 32-bit instruction words (Figure 3b) consist of a 12-bit operation code and a 20-bit address part. The address part is encoded in the residue code with the check modulus 15. An address part consists of a 16-bit binary address a and a 4-bit check symbol c(a). The check symbol c(a) has the value

$$c(a) = 15 - |a|_{15}$$

where $|a|_{15}$ is the modulo 15 residue of a. The checking algorithm computes the modulo 15 residue of an address and adds it (modulo 15) to the check symbol c(a).

A zero sum (represented by 1111) indicates a properly coded address part. The residue code is preferable for address parts over the product code because it is separable, and the address a is available to the memory address decoding circuits in its ordinary binary form. It is important to note that the "one's complement" $15 - |a|_{15}$ rather than the residue $|a|_{15}$ itself is used as the check symbol c(a). In this case the fault-detection effectiveness in byte-serial operation remains the same as for product-coded operands, while the use of $|a|_{15}$ as a check symbol gives a lower effectiveness. Furthermore, the bus checking algorithm is the same for product-coded operands and for address parts—it is simply a modulo 15 summation of all bytes and a test of the result for the zero value represented by 1111.

The *operation code* is divided into three bytes of four bits each. The operation code bytes are protected by a 2-out-of-4 encoding, which leaves six valid words in a four-bit byte. Such coding is most efficient for short words and is acceptable in a computer because operation codes are not subjected to arithmetic operations. It is evident that their validity must be tested by a separate checking circuit, since it cannot be verified by the modulo 15 checker (which is bypassed by the operation codes). The separation of the operation code into three separately encoded bytes facilitates the decoding and validity testing of op. codes received by the functional units. The 2-out-of-4 encoding gives a total of 216 distinct combinations for operation codes. The indication of index registers which are to be used

must be contained within the operation code. Since the Model I contains two index registers, every indexable operation code requires three distinct combinations, while non-indexable operations require only one each.

It is apparent that residue encoding with the check modulus 15 can be applied to the numeric operands and to the operation codes as well as to the address parts. Such use of a uniform residue code has the advantages of code separability and of identical check algorithms. In the case of operation codes, a modulo 15 residue-coded 12-bit number provides 256 distinct combinations. On the other hand, the 2-out-of-4 coding for individual bytes of the operation code permits validation and use of individual bytes. The choice of three different methods of encoding in the Model I was motivated by the need to gain detailed insight into their relative merits and shortcomings.

One instruction cycle is executed by the Model I JPL-STAR system in three steps. In the first step, the address of the instruction is sent from the Sequence Register in the Control Arithmetic Processor to the memory units; the transmission of the address is tested by the Bus Checker. In the second step, the addressed memory unit broadcasts the operation code and address to all functional units. The appropriate units recognize the code, accept the address, and initiate execution. In the third step (if needed) the instruction is executed, a result is placed on the bus and accepted by the destination unit. The Bus Checkers test every word on the busses for proper encoding.

Functional units of model I

The replaceable functional units of the computer have a standard format for their output words and have the same internal communication lines, as shown in Figure 1 and described previously. The Input/Output and Interrupt units also have external communication lines to the spacecraft. Brief descriptions of the functional units are given in this section.

The Main Arithmetic Processor (MAP) accepts the operands and delivers the results as 32-bit product-coded binary numbers. All arithmetic control is contained in the MAP; an input consists of an operation code (add, subtract, multiply, divide) followed by a coded operand, and the output is a coded result followed by a 2-out-of-4 Condition Code byte, indicating either one of three singularities (sum overflow, quotient overflow, zero divisor) or the type of a good result (positive, zero, negative). The good result codes are used by the CAP as data for conditional jump instructions. All partial and final results are delivered to the Bus Checker #2. A breadboard model of the MAP has been constructed and tested.[46] It is believed to be the first complete arithmetic processor for product-coded operands.

The Control Arithmetic Processor (CAP) contains two Index Registers (IR), the Sequence Register (SR), the Condition Code Register (CCR), and an adder. When the 16-bit index word x from the IR is added to an address a, its 4-bit check symbol c(x) is added modulo 15 to c(a). The indexed address and the new check symbol go past the Bus Checker #2 to the input lines of the memory units. The incrementing (by one) of the current address s in the SR is performed in exactly the same manner, with 1 added to s and 14 = c(1) added modulo 15 to c(s). The incremented address is returned to the SR. The presence of the Condition Code byte in the CAP permits fast execution of conditional jump instructions.

The Logic Processor (LOP) performs the bit-by-bit logic operations, shifts, and code conversions on input operand words. It contains the Logic Accumulator (LAR) and the Mask Register (MR). The arithmetic coding is removed from the operand before the operation, since arithmetic codes are not preserved during logic operations. The final result is again encoded; provisions exist for both product and residue codes. The LOP is therefore capable of encoding input words, removing code from output words, and executing conversions between product and residue codes. A four-bit adder is included for encoding and decoding operations. Functioning of the LOP is checked by operating two copies concurrently and comparing all results.

The Read-Only Memory (ROM) unit contains the permanent programs and the associated constants for a given mission. The experimental model provides 2^{14} words of 32 bits each, using a "braid" assembly of transformers and wires for the permanent storage of binary information.[47] The ROM also contains all necessary peripheral electronics: the op. code, address, and output registers, access circuits, drivers, sequence control, and monitoring circuits. All output words from the ROM are delivered past Bus Checker #1. The Model I JPL-STAR computer includes complete replicas of the ROM as replacements; the replacement of peripheral electronics without discarding the core and wire assembly is being studied.

The Read-Write Memory (RWM) units are self-contained 4096 word modules with the same peripheral electronics as the ROM. Direct addressing is provided for 64K words, including the 16K ROM; this permits up to 12 RWM units. Each RWM unit has three modes of operation. In the *standard* mode a RWM unit recognizes its own wired-in *unit address,* and is connected to both input and output busses. In the *auxiliary* mode, a RWM unit stores and assumes the unit address of another unit, to be called its *main* unit. The

auxiliary unit stores the same input words as its main unit. When the main unit reads out a word to the bus, the auxiliary unit reads out the same word internally and compares it to the word which is on the bus. If the words disagree, the auxiliary unit signals a comparison error to the CDU. A "reverse" diagnostic command causes a reversal of roles between an auxiliary unit and its main unit. The third mode of RWM operation is the *relocated* mode, which utilizes a stored unit address, but otherwise is identical to the standard mode. A RWM unit is placed into or released from the auxiliary or relocated mode by special instructions which are directed to its wired-in unit address. The auxiliary mode permits a redundant storage of machine words in one, two, or more separate units with continued comparison of readout supplementing the Bus Checker. The relocated mode permits a replacement of a failed RWM unit by a spare with the same unit address. The choice of employing redundant storage is left to the user.

The Interrupt Unit (IRU) and the Input-Output Unit (IOU) serve as interfaces with the external world. The IOU contains buffer registers for receiving and delivering machine words. The IRU receives commands from an outside operator and service requests from other parts of the spacecraft system. An interrupt is effected when the IRU places a properly coded instruction word on the bus, preempting the delivery of the next instruction specified by the Sequence Register. The details of interface protection remain to be established for specific spacecraft systems; in general, complete duplication will serve under most conditions.

The System Clock Unit (SCU) contains counters needed for the sequencing and timekeeping functions of the computer and the spacecraft. Counter settings are coded machine words. The SCU generates an internal interrupt request when a preset count has been reached.

The two Bus Checkers (CH1 and CH2) serve to check all machine words which are being transmitted on the busses for validity of encoding. The checking of arithmetic codes requires a four-bit Check Sum Accumulator (CSA) and a four-bit modulo 15 adder (with an end-around carry) which adds the bytes being transmitted to the word in the CSA. An error-status line to the CDU indicates whether the CSA contains an acceptable word (1111). The checking of the non-numeric 2-out-of-4 operation code bytes is carried out by a separate logic circuit which also has an error-status line to the CDU. The relatively small size of the Bus Checkers makes their duplication quite practical for fault-tolerance.

The Control-Diagnosis Unit (CDU) issues the control signals for timing and for replacement. It also initiates the recovery actions when a fault is indicated by status outputs of the powered functional units and bus checkers. In normal operation, the internal timing of the units is initiated for every instruction with the "Sync"signal and is tested by reception of the "Complete" signal. A copy of the current instruction is stored in the CIR register within the CDU and provides the data to verify all "Active" and "Complete" signals and the Bus Checker status signals. The CDU also contains the Rollback Point Register (RPR) which is loaded with an address under program control. When a fault is detected from the status signals, computing is "rolled back" to the instruction at this address and repeated in order to correct a transient fault. If a persistent fault is indicated by the same unit, a replacement is carried out, and the program is resumed at the rollback point. Software diagnosis may be employed to provide additional data on permanent faults. Periodic updating of the RPR is the responsibility of the programmer. For catastrophic transient faults (e.g., brief power loss) the CDU contains a wired-in "Restart" procedure.

The CDU acts as the "hard core" of the system and requires immediate fault-tolerance. The Model I STAR computer maintains four powered CDU copies. The CDU outputs are determined by a majority vote of three units; in the case of a disagreement, the minority unit is at once replaced by the operating fourth unit, and a new spare CDU is brought in and synchronized with the other powered units to act as an operating spare. Because of the four unit requirement, design effort has been concentrated on reducing the CDU to the least possible complexity. Experience with Model I is expected to yield further insights into this problem and to lead to modifications of the CDU design.

The performance of the JPL-STAR computer must be measured against an equivalent non-redundant computer. The important parameters are complexity, gain in mean life, and types of faults which can be tolerated. The first design objective has been the ability to tolerate a wide range of faults, including catastrophic (multiple dependent) transient and permanent faults in the functional units. The gain in mean life is being investigated by means of simulation (hardware and software) as well as by mathematical methods. The results of these studies will be reported upon completion of construction and hardware simulation.

ACKNOWLEDGMENT

The research described in this paper has been carried out at the Jet Propulsion Laboratory, Pasadena, California, under Contract NAS7-100, sponsored by the National Aeronautics and Space Administration. The author wishes to acknowledge the support and encouragement of W. F. Scott and discussions with J. J. Wedel and G. R. Hansen. System and logic design of

several functional units was performed by D. A. Rennels and A. D. Weeks, all of the Flight Computers and Sequencers Section, Guidance and Control Division, JPL.

REFERENCES

1 J VON NEUMANN
Probabilistic logics and the synthesis of reliable organisms from unreliable components
Automata Studies C E Shannon and J McCarthy eds Annals of Math Studies No 34 Princeton University Press 1956 p 43-98

2 C J CREVELING
Increasing the reliability of electronic equipment by the use of redundant circuits
Proceedings of the IRE 44 509-515 April 1956

3 E F MOORE C E SHANNON
Reliable circuits using less reliable relays
Journal of the Franklin Institute 262 Nos 9 and 10 191-208 and 281-297 Sept Oct 1956

4 T B LEWIS
Primary processor and data storage equipment for the orbiting astronomical observatory
IEEE Transactions on Electronic Computers EC-12 No 5 677-686 Dec 1963

5 W G BROWN J TIERNEY R WASSERMAN
Improvement of electronic computer reliability through the use of redundancy
IRE Transactions on Electronic Computers EC-10 No 3 407-416 Sept 1961

6 M M DICKINSON J B JACKSON G C RANDA
Saturn V launch vehicle digital computer and data adapter
AFIPS Conference Proceedings 26 501-516 Fall JCC 1964

7 R H URBANO
Some new results on the convergence oscillation and reliability of polyfunctional nets
IEEE Trans on Electronic Computers EC-14 No 6 769-781 Dec 1965

8 J G TRYON
Quadded logic
Redundancy Techniques for Computing Systems Spartan Press Inc Washington DC 1962 p 205-228

9 W H PIERCE
Failure-Tolerant Computer Design
Academic Press Inc New York 1965

10 S WINOGRAD J D COWAN
Reliable Computation in the Presence of Noise
The MIT Press Cambridge Mass 1963

11 J J DONEGAN C PACKARD P PASHBY
Experiences with the Goddard computing system during manned spaceflight missions
Proceedings of the 19th National Conference of the ACM p A2.1-1 to A2.1-8 1964

12 J E HAMLIN
A general description of the NASA real time computing complex
Proceedings of the 19th National Conference of the ACM p A2.2-1 to A2.2-22 1964

13 R T STEVENS
Norad's computers get all the facts
Electronics 40 No 4 p 113-118 Feb 20 1967

14 W W PETERSON
Error correcting codes
The MIT Press and John Wiley and Sons Inc New York 1961

15 W H KAUTZ
Codes and coding circuitry for automatic error correction within digital systems
Redundancy Techniques for Computing Systems Spartan Press Inc Washington DC 1962 p 152-195

16 A AVIZIENIS
A study of the effectiveness of fault-detecting codes for binary arithmetic
JPL Technical Report No 32-711 Jet Propulsion Laboratory Pasadena Calif Sept 1 1965

17 A AVIZIENIS
Concurrent diagnosis of arithmetic processors
Paper No 3.1 Conference Digest of the 1st Annual IEEE Computer Conference Chicago Illinois Sept 6-8 1967

18 D B ARMSTRONG
A general method of applying error correction to synchronous digital systems
Bell System Tech Journal 40 No 2 577-594 March 1961

19 S SESHU D N FREEMAN
The diagnosis of asynchronous sequential switching systems
IRE Transactions on Electronic Computers EC-11 No 4 459-465 August 1962

20 J P ROTH
Diagnosis of automata failures: A calculus and a method
IBM Journal of Research and Development 10 No 4 278-291 July 1966

21 W C CARTER H C MONTGOMERY R J PREISS H J REINHEIMER
Design of serviceability features for the IBM System/360
IBM Journal of Research and Development 8 No 2 115-126 April 1964

22 R E FORBES D H RUTHERFORD C B STIEGLITZ L H TUNG
A self-diagnosable computer
AFIPS Conference Proceedings 27 Part 1 1073-1086 Fall JCC 1965

23 E MANNING
On computer self-diagnosis: Part I—Experimental study of a processor Part II—Generalizations and design principles
IEEE Trans on Electronic Computers EC-15 No 6 873-890 Dec 1966

24 R W DOWNING J S NOWAK L S TUOMENOKSA
No 1 ESS maintenance plan
The Bell System Technical Journal 43 No 5 Part 1 1961-2019 Sept 1964

25 B J FLEHINGER
Reliability improvement through redundancy at various system levels
IBM Journal of Research and Development 2 No2 148-158 April 1958

26 J E GRIESMER R E MILLER J P ROTH
The design of digital circuits to eliminate catastrophic failures
Redundancy Techniques for Computing Systems Spartan Press Inc Washington D C 1962 p328-348

27 A AVIZIENIS
Coding of information for a guidance computer with active redundancy
JPL Space Programs Summary No 37-22 Vol IV 9-12 Jet Propulsion Laboratory Pasadena Calif 1963

28 D E ROSENHEIM R S ASH
Increasing reliability by use of redundant machines
IRE Trans on Electronic Computers EC-8 125-130 June 1959

29 R TEOSTE
Design of a repairable redundant computer
IRE Trans on Electronic Computers EC-11 No 5 643-649 October 1962

30 J KRUUS
Upper bounds for the mean life of self-repairing systems
Coordinated Science Laboratory R-172 University of Illinois Urbana Illinois July 1963 AD-418 174

31 I TERRIS M A MELKANOFF
Investigation and simulation of a self-repairing digital computer
IEEE Conference Record on Switching Circuit Theory and Logical Design 264-272 1965

32 R L ALONSO A L HOPKINS JR H A THALER
A multiprocessing structure
Paper No 6.2 Conference Digest of the 1st Annual IEEE Computer Conference Chicago Illinois Sept 6-8 1967

33 J GOLDBERG K N LEVITT R A SHORT
Techniques for the realization of ultra-reliable spaceborne computers
Final Report—Phase 1 Contract NAS12-33 Stanford Research Institute,Menlo Park Calif Sept 1966

34 R E LYONS W VANDERKULK
The use of triple-modular redundancy to improve computer reliability
IBM Journal of Research and Development 6 No 2 200-209 April 1962

35 H L ERGOTT D P ROSENBERG
Modular redundancy for spaceborne computers
Proceedings of IEEE Spaceborne Computer Engineering Conference 137-150 Anaheim California Oct 30-31 1962

36 K J GURZI
Estimates for best placement of voters in a triplicated logic network
Trans IEEE EC-14 No 5 711-716 October 1965

37 D K RUBIN
The approximate reliability of triply redundant majority-voted systems
Paper No 3.4 Conference Digest of the 1st Annual IEEE Computer Conference Chicago Illinois Sept 6-8 1967

38 J E ANDERSON F J MACRI
Multiple redundancy applications in a computer
Proc 1967 Annual Symposium on Reliability 553-562 Washington D C Jan 10-12 1967

39 M A BREUER
General survey of design automation of digital computers
Proc of the IEEE 54 1708-1721 Dec 1966

40 W G BOURICIUS W C CARTER J P ROTH P R SCHNEIDER
Investigations in the design of an automatically repaired computer
Paper No 6.4 Conference Digest of the 1st Annual IEEE Computer Conference Chicago Illinois Sept 6-8 1967

41 P W AGNEW R E FORBES C B STIEGLITZ
An approach to selfrepairing computers
Paper No 6.3 Conference Digest of the 1st Annual IEEE Computer Conference Chicago Illinois Sept 6-8 1967

42 D EMULLER
Evaluation of logical and organizational methods for improving the reliability and availability of a computer
Paper No 6.1 Conference Digest of the 1st Annual IEEE Computer Conference Chicago Illinois Sept 6-8 1967

43 E K VAN de RIET D R BENNION J M YARBOROUGH
Feasibility study for a reliable magnetic connection switch
Final Report—Phase I Contract 951232 under NAS7-100 Stanford Research Institute Menlo Park Calif Feb 1966

44 D T BROWN
Error detecting and correcting codes for arithmetic operations
IRE Trans EC-9 333-337 1960

45 H L GARNER
Error codes for arithmetic operations
IEEE Trans EC-15 763-770 1966

46 A AVIZIENIS
The diagnosable arithmetic processor
JPL Space Programs Summary 37-37 IV Jet Propulsion Laboratory Pasadena Calif 1966 76-80

47 W H ALDRICH R L ALONSO
The 'braid' transformer memory
IEEE Trans EC-15 502-508 August 1966

Some relationships between failure detection probability and computer system reliability

by HENRY WYLE and GERALD J. BURNETT
Autonetics
Anaheim, California

A. INTRODUCTION

The relationships between computer failure rates and the failure rates of the modules from which the computers are constructed are well known. The analytical techniques permitting derivation of one parameter from the other for a given design are in widespread use. With the increasing interest in ultra-reliable computer systems various approaches to increasing reliability through the use of redundancy have been proposed and in some cases implemented. A feature common to many of these approaches is the inclusion of on-line spare modules (either used or idle) with provision in the computer system for automatic replacement of a failed module by an on-line spare. Systems of the "graceful degradation" type generally fall into this class, as well as some "self-repairing" systems and other system types. Such systems are basically self-reconfiguring. Using modules of ordinary failure rates they are theoretically capable of astronomically high reliabilities. There is to data, however, a good deal of reserve on the part of the general computing community about accepting these reliabilities at face value since theory and practice are usually separated from each other by a host of difficult and sometimes ill-defined problems.

In the case of reconfigurable computer systems, the ability of a system to reconfigure itself in response to a module failure cannot be guaranteed, principally for two reasons. First of these is the existence of critical failure points. The number of critical failure points and the design difficulty in identifying and minimizing them are functions of the type and complexity of the system design. This obstacle to guaranteed reconfigurability promises to remain a problem area for some time, since its solution seems primarily to be a matter of inspired design in specific systems rather than being amenable to analysis methods. The second obstacle is the acknowledged imperfection of failure detection methods. Repertoire and completeness of failure detection techniques available to the computer designer are growing rapidly, but each technique carries a price tag. Thus, the reconfigurable computer system designer's problem can be stated as: for the system approach I have chosen, what is the optimum balance between failure detection completeness and module reliability to achieve my system reliability goal? The answer to this question for a specific system design would provide specifications for the hardware and failure detection designers to work toward. The answer to this question for a variety of system designs would provide the system designer with greater insight into the advantages of one system approach over another. Thus development of analytical techniques relating the completeness of failure detection (or Probability of Failure Detection, P_d) in each module of a system to the overall reliability (or Probability of Success, PS) of the system would seem to be an important step toward the realization of better reconfigurable computer systems.

In the course of a design study toward a fail-safe multi-processor for an aerospace application, a preliminary design emerged which had a theoretical probability of critical computation survival (PCCS) of 0.99986, assuming perfect failure detection. To observe the effect of imperfect failure detection on the PCCS, the parameter, P_d, was introduced into the reliability calculations and it was discovered that the PCCS of the particular multiprocessor under study was more critically dependent upon the P_d of its modules than upon their MTBF. This effect appeared to warrant further investigation. It was gradually learned that the limitation by P_d of system survival (or availability) applies to multi-processors, multiple-computers, and single computers supported by a regular maintenance program. Furthermore, there would appear to be no inherent limitation to the field of com-

puters of the methods applied in this design study. Rather, these methods, being extremely simple, appear to be generally applicable to the design and evaluation of systems in which equipment failures may be tolerated because appropriate corrective action is thought to be possible.

This report will develop quantitative relationships between P_d and PS. A simplified model, the general calculation methods, and some results are described in Sections B and C. The method of applying this analysis technique to asymmetrical multiprocessor configurations is shown in Section D. The applicability of the technique to a single computer with off-line, spare modules is indicated in Section E.

B. The model

The model selected for this study consists of a number of identical equipment modules, of which at least some smaller number must survive for an interval of time (a mission of some specific duration) without failures to permit system survival. It is assumed that all modules are known to be operational at the start of the mission, and that no repair of failed modules is possible for the duration of the mission. When modules fail, they are assumed to do so one at a time. It is further assumed that any subset of the original number of modules may permit the system to survive provided the subset is sufficiently large.

The selected model is intended to have fairly broad applicability in its own right and can also be used to complement models assuming repair during the mission in that it can describe the behavior of the systems until the repair is effected.

One further aspect of the model remains to be defined, i.e., the criteria by which the system is said to have failed. The first criterion for system failure is, of course, that fewer modules than some specified minimum remain operable during the mission. Logically, the second criterion for system failure should be that a computing module performing a system-essential function has failed undetectably. It can also be argued that any computing module failure which goes undetected, even a computing module engaged for the moment solely in auxiliary system functions, could cause a system failure and should therefore be considered as the second criterion for system failure. This criterion is more stringent than its alternate and its appropriateness depends upon the software and hardware designs of the multi-module system under consideration. The current study has included both of these alternate "undetectability" criteria for system failure. Only the more stringent form of the criterion will be used here since the calculations employing the alternate form are not yet complete; thus, the second criterion for system failure employed in the current model is that an undetected failure occur in any module of the system.

The next step in the study consisted of selecting various parameters of the model and in perturbing these parameters to obtain useful conclusions in terms of various optimization criteria. With this step the generality of the model tends to diminish since the parameter values are so highly dependent upon the proposed system's environment, its performance requirements, and the economics of its hardware technology. Thus, the conclusions drawn from the parameter perturbations used may or may not be applicable to broad classes of systems. However, the methods used retain their generality and the individual designer may use those parameter values which are most appropriate to his needs.

Parameters of the model are as follows:

- n = total number of modules in the system
- m = number of unfailed modules needed for system survival
- p_f = probability of failure of each module some time during the mission. This parameter thus includes both the mission duration and the module MTBF.
- p_{nd} = probability of not detecting an occurred module failure
- PS = probability of system survival throughout the mission
- $P_F = 1 - PS$ = probability of system failure during the mission
- n/m = redundancy factor in initial system.

Assuming all modules to be identical and symmetrically configured in the system, the probability of system failure is the sum of two summations: (1) more than (n-m) modules failing and (2) any modules failing undetectably. This sum is:

$$P_F = \sum_{k=(n-m+1)}^{k=n} \frac{n!}{(n-k)!\,k!} (p_f)^k (1 - p_f)^{n-k}$$

$$+ \sum_{k=1}^{k=(n-m)} \frac{n!}{(n-k)!\,k!} (p_f)^k (1 - p_f)^{n-k} \left[1 - (1-p_{nd})^k\right] \quad (1)$$

This equation is used to generate the curves shown in Section C. The equation will be generalized to take asymmetry and non-identity of modules into account in Section D.

System design considerations and experimental perturbations of parameters

Some of the questions which typically confront the designers of highly reliable, multi-module com-

puting systems are listed here and an attempt made to consider them in the light of p_{nd}. This leads to calculations in which an attempt is made to gain insight into the effect of p_{nd} on the optimum balance of the other parameters of the system:

1. Is there a p_{nd} "threshold" and how is its value affected by the other parameters of the system?
2. Is it better to have many small units or a few large units?
3. How much redundancy should the system contain?

1. Effects of system parameters on p_{nd} threshold

Suppose that a unit computing module of a given MTBF (and therefore having a given p_f) is available. Suppose further that a given redundancy factor has been deemed advisable. Thus, if a range of computing requirements is considered, greater or fewer computing units will be required. Figure 1 shows P_F as a function of p_{nd} for three equipment configurations. In all three cases, the redundancy factor, n/m, is 2. In all three cases the p_f of each equipment module is 0.3 The figure shows the effect of p_{nd} in reversing the system reliability relationships of the three configurations: for low p_{nd}, the 8-unit system is more reliable; for high p_{nd}, the 2-unit system is more reliable. This effect becomes more pronounced as p_f decreases from the rather high value of 0.3 to more typical avionics equipment values.

In Figure 2, the different effects appearing in Figure 1 have been somewhat separated. Figure 2 shows the effect of an increasing p_{nd} in increasing the P_F of the two extreme equipment configurations of Figure 1. The P_F for the two configurations have been normalized to 0 at their minimum values, and to 1.0 at their maximum values. Thus, Figure 2 isolates the changing slope of P_F (p_{nd}) as a parametric function of the total amount of equipment. It is seen that the system with less equipment, (n = 2, m = 1), has a slightly more pronounced threshold effect with regard to p_{nd}, i.e., it tends to be slightly more tolerant of a poor p_{nd} at low p_{nd} values, but then climbs more rapidly than the other system toward its particular P_F max once p_{nd} exceeds about 0.13. This effect holds for all values of p_f, although the threshold values of p_{nd} and the relative importance to the system's P_F differ.

2. p_{nd} and optimum module size

Suppose that a specific computing requirement and a redundancy factor have been established. One of the major system design considerations is often whether to have many small modules or a few big ones, the total performance capability being the same for either case. To shed light on the effect of p_{nd} on the question, a diagram similar to Figure 1 would be appropriate, with one difference, i.e., instead of using the same p_f for the unit equipments of all three systems, the p_f of the unit equipments should differ in accordance with the lower failure rates of modules in systems containing many small modules as opposed to a few big modules. To achieve the same computing capability with half as many modules requires an increase in equipment complexity which is very much a function of the technology employed, the capabilities in question, and various other factors. In Figure 3, the curves of Figure 1 are redrawn, but with the assumption that each doubling in module capability results in a 20 percent increase in module complexity (and failure rate). Figure 4 assumes that a doubling in module capability results in a 50 percent increase in module complexity.

A comparison of the curves of Figures 3 and 4 shows the importance of the complexity ratio assumed for increasing module capability. Figure 3 gives the advantage to the four-module system only for p_{nd} of 0.1 or less, whereas Figure 4 shows the four-module system to be more reliable all the way to a p_{nd} of 0.25, a very generous p_{nd} by current standards.

In Figures 5a and 5b, a module complexity increase of 41 percent per doubling of module capability has been assumed. The difference between the two figures is in the reference module p_f's used. These figures thus illustrate the effect of the absolute p_f in shifting the trade-off between many small modules and a few big modules. Among the effects which may be noted are the slightly greater criticality of p_{nd} in the systems with lower module p_f's, and the shift in the p_{nd} crossover point from $p_{nd} = 0.023$ in the low p_f system to $p_{nd} = 0.086$ in the high p_f system. Note also that at a fixed, low value of p_{nd}, say 0.05, the failure probability P_F of the eight-module system, varies linearly with the p_f of its modules, whereas the p_f of the two-module system does not.

Again it should be recalled that these comparisons neglect the critical failure point aspects of the system design so that the final reliability comparison cannot easily be generalized and cannot be achieved by this analysis alone. Another point which should be kept in mind is that the complexity ratio valid for a particular system is a function of the system design as much as it is a function of the hardware technology employed. The "overhead" of memory capacity and logical circuitry required for each module is very much a function of the simplicity of the switching scheme, computing algorithms, executive approach, and the other aspects of the system design.

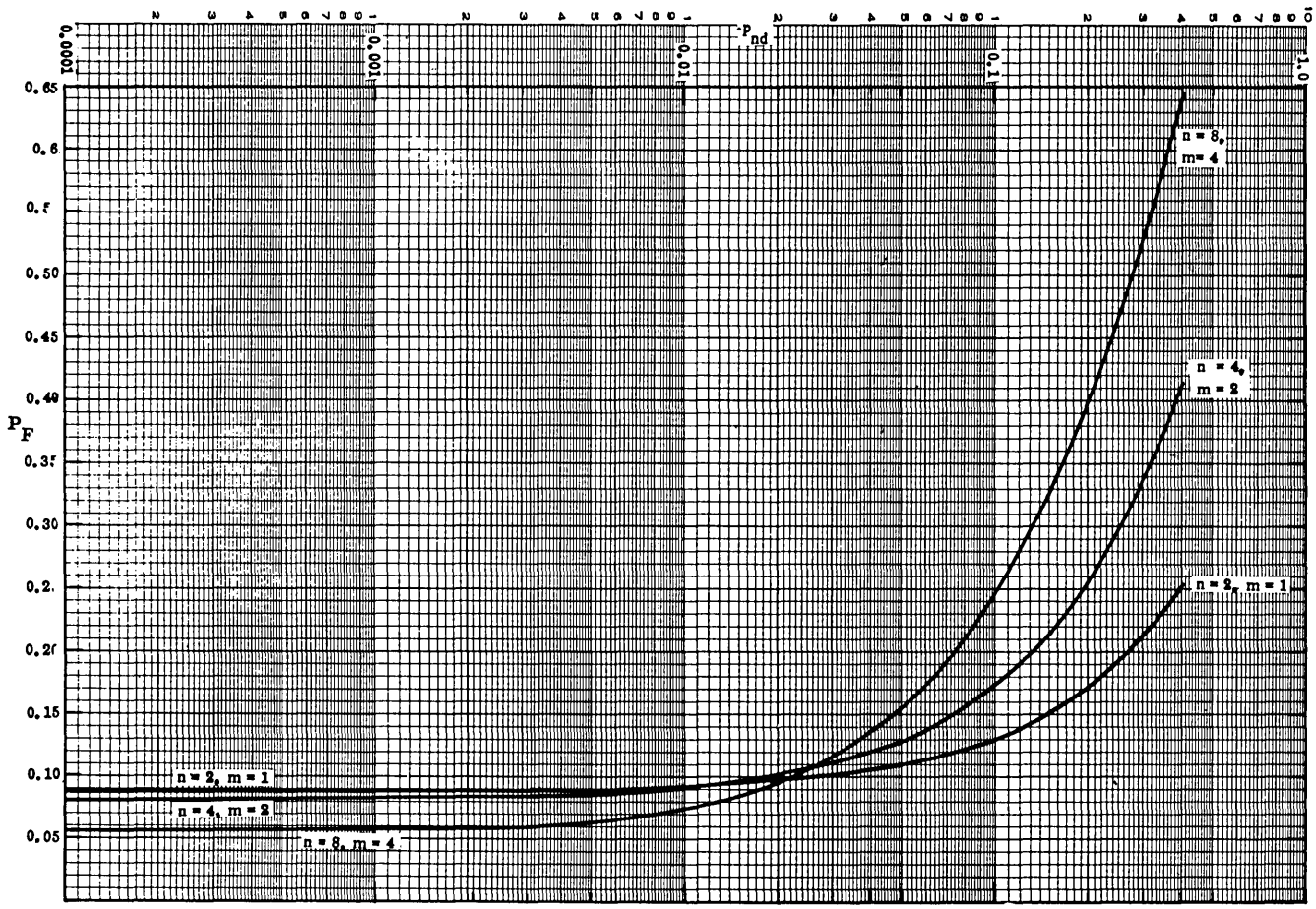


Figure 1 – Effect on P_F (p_{nd}) of amount of required equipment.
. $p_f = 0.3$. n/m = 2.

3. How much redundancy should the system contain?

Suppose that a specific computing requirement and an over-all reliability requirement have been established. One of the system design decisions is the determination of the optimum number of redundant modules by means of which the required system reliability is to be reached. It will be seen that p_{nd} as well as p_f must be considered in determining the most economical design.

Assume that two molecules are required for the system-essential computations. In Figure 6, a three-module, a four-module, and a six-module system are characterized, thereby comparing systems with redundancy factors of 1.5, 2 and 3. The modules in Figure 6 are assumed to have a p_f of 0.2. In Figure 7 the same comparisons are made, but with modules p_f's of 0.025.

Figures 6 and 7 show an important phenomenon. Depending upon the attainable p_f and p_{nd}, the theoretical reliability of a multi-module computing system may be degraded by adding more than a minimal amount of redundancy. For example, for $p_f = 0.025$ (Figure 7), it is more reliable to have only one spare module rather than two or four, for a typical current-day p_{nd} such as 0.075. Even for a p_{nd} as low as 0.03 (a very difficult p_{nd} to achieve in a computer), the improvement obtained in system reliability by adding a second spare unit to the system is minor. This is in contrast to the conclusion resulting from neglect of p_{nd}. That conclusion would be that adding a second spare unit would increase the reliability of the system by a factor of 30.

The theoretical results illustrated in Figures 3 through 7 may be qualitatively summarized as follows:

1. Adding more on-line spare modules to a reconfigurable computer system to increase its redundancy ratio is not as effective in increasing system reliability as desired because imperfect failure detection decreases system reliability as the amount of hardware in the system increases.

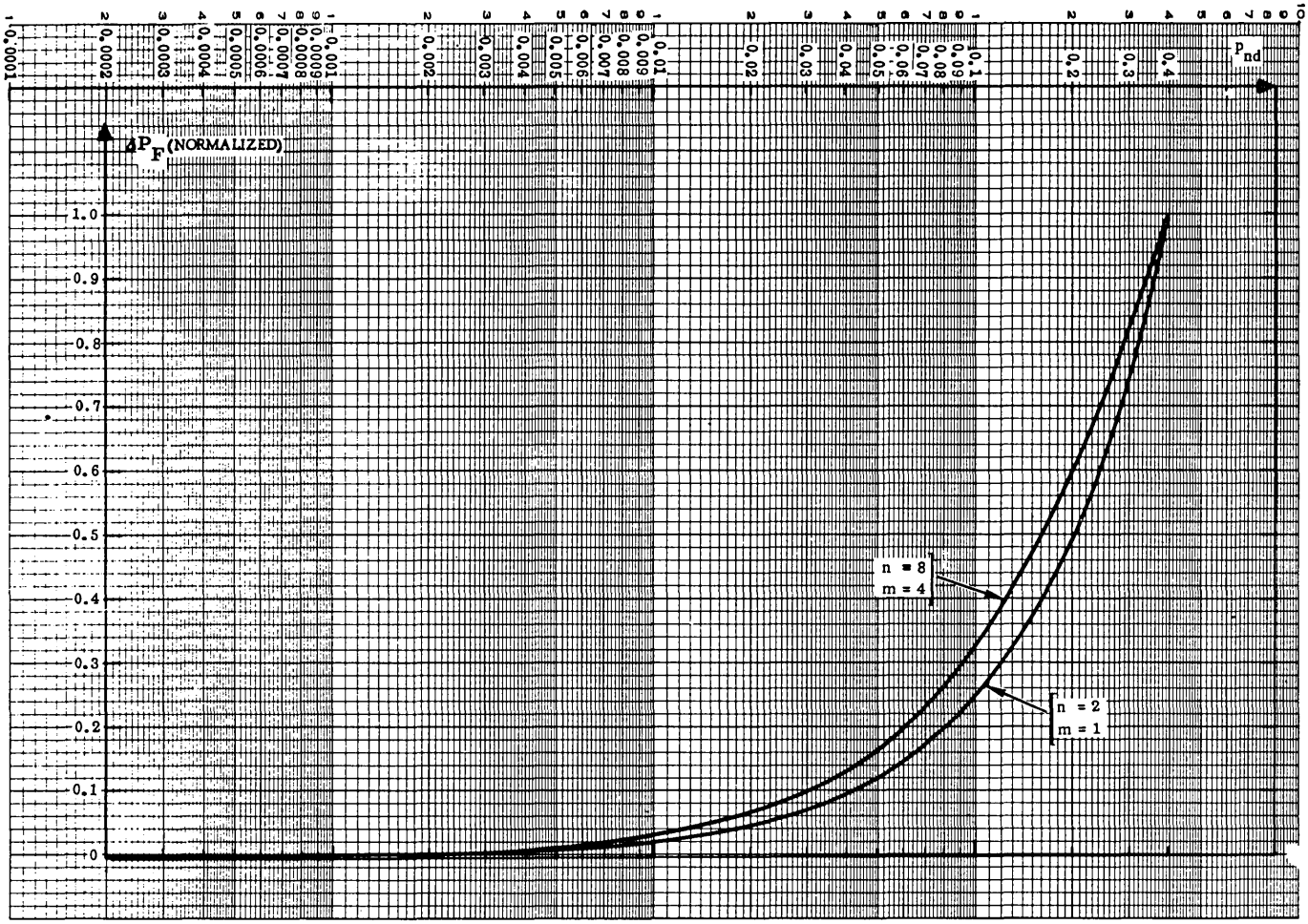


Figure 2 – Effect on p_{nd} threshold of amount of required equipment.
. p_f=0.3.n/m=2.

2. Reducing the complexity of the unit module and having more of them in a reconfigurable computer system with a given redundancy ratio in order to be capable of tolerating a greater number of equipment failures can similarly have disappointing system reliability results because of the effects of imperfect failure detection.

The conclusions are further supported by the fact that the present analysis assumed no switching overhead and neglected the effects of critical failure points in degrading system reliability. Systems containing a greater number of modules which must work cooperatively tend to have a greater number of critical failure points and require more switching circuitry and other forms of overhead than do systems containing fewer modules. This in turn leads to more hardware and still further reliability degradation.

D. Systems containing non-identical modules

The equation given in an earlier section for calculating P_F of a multi-processor configuration assumed all modules to be identical. This assumption is in most cases not valid, and it is therefore desirable to generalize the equation to apply to multi-processors whose modules play different roles in the system and are also physically different from each other. An example of such a multi-processor is one containing, m memory modules; n processor modules; and q intercommunication and I/O control modules. Only a subset, m_o, of the m memory modules need be operable in order for the system to survive. Similarly, only subsets n_o and q_o of the other two module types are required for system survival. We further define:

P_F / = probability that a memory module will fail sometime during the mission

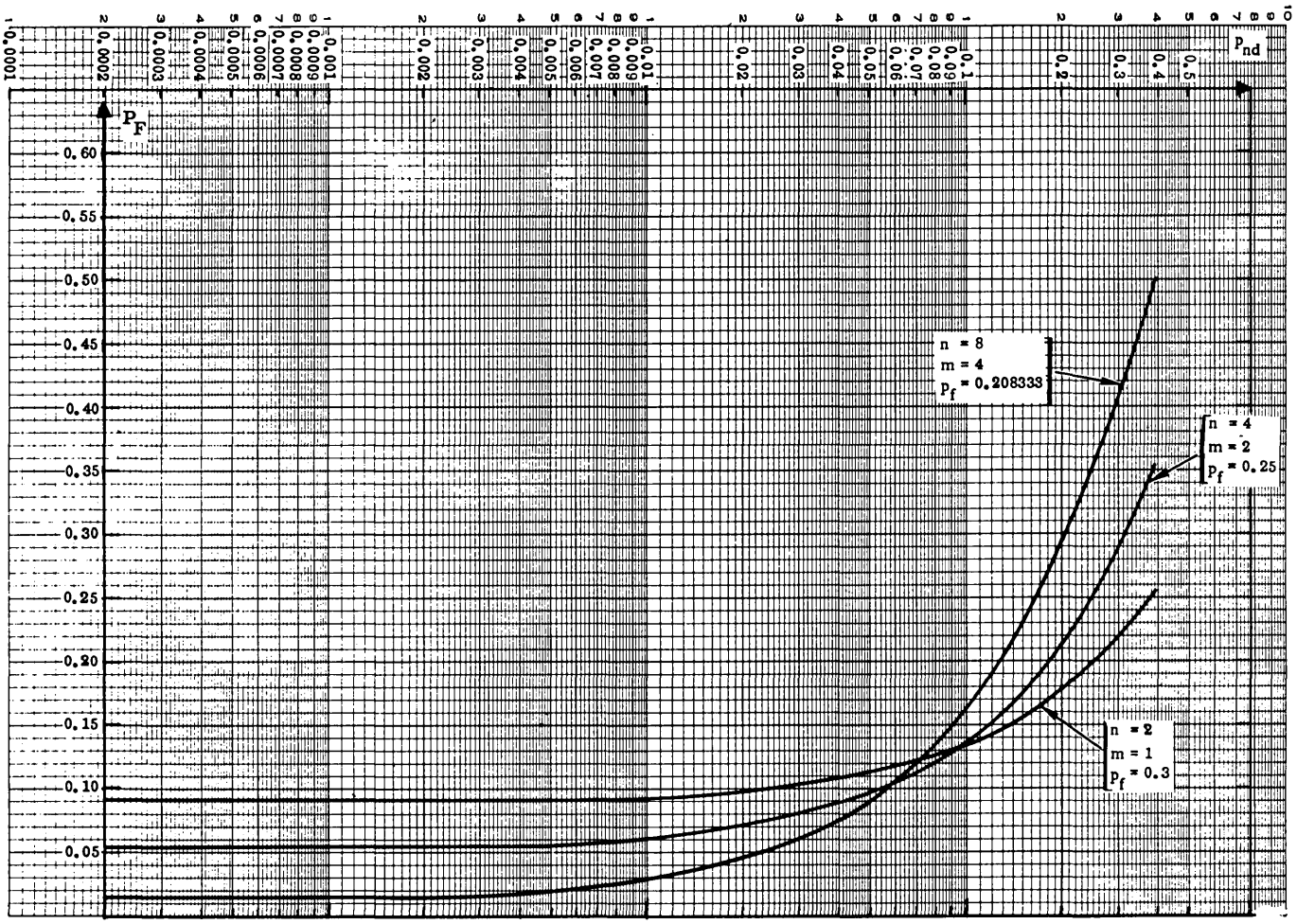


Figure 3—Effect on P_F (p_{nd}) of module size, assuming fixed total computing capability. Doubling module capability is assumed to increase module complexity by 20 percent.

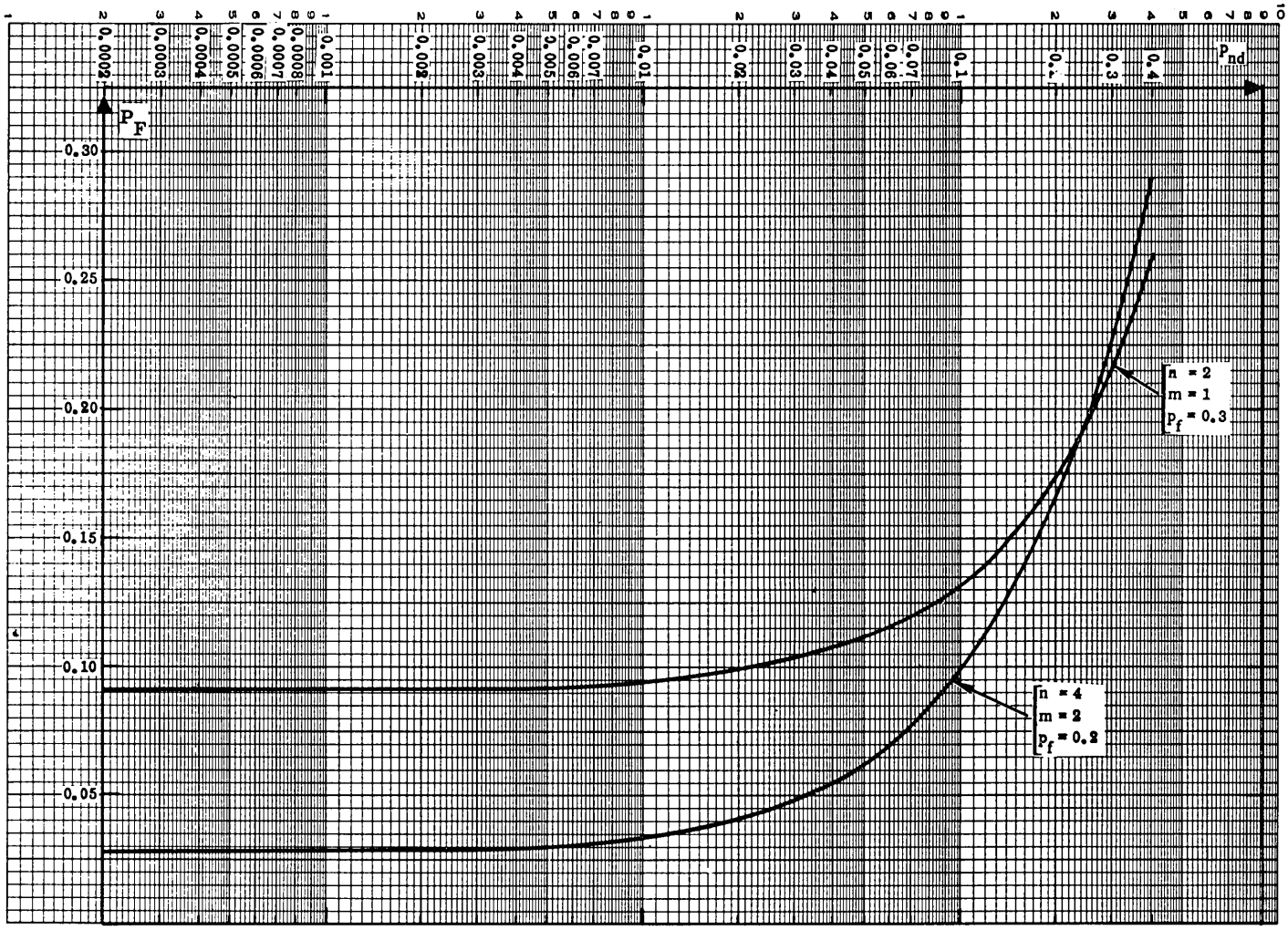


Figure 4—Effect on P_F (p_{nd}) of module size, assuming fixed total computing capability. Doubling module capability is assumed to increase module complexity of 50 percent

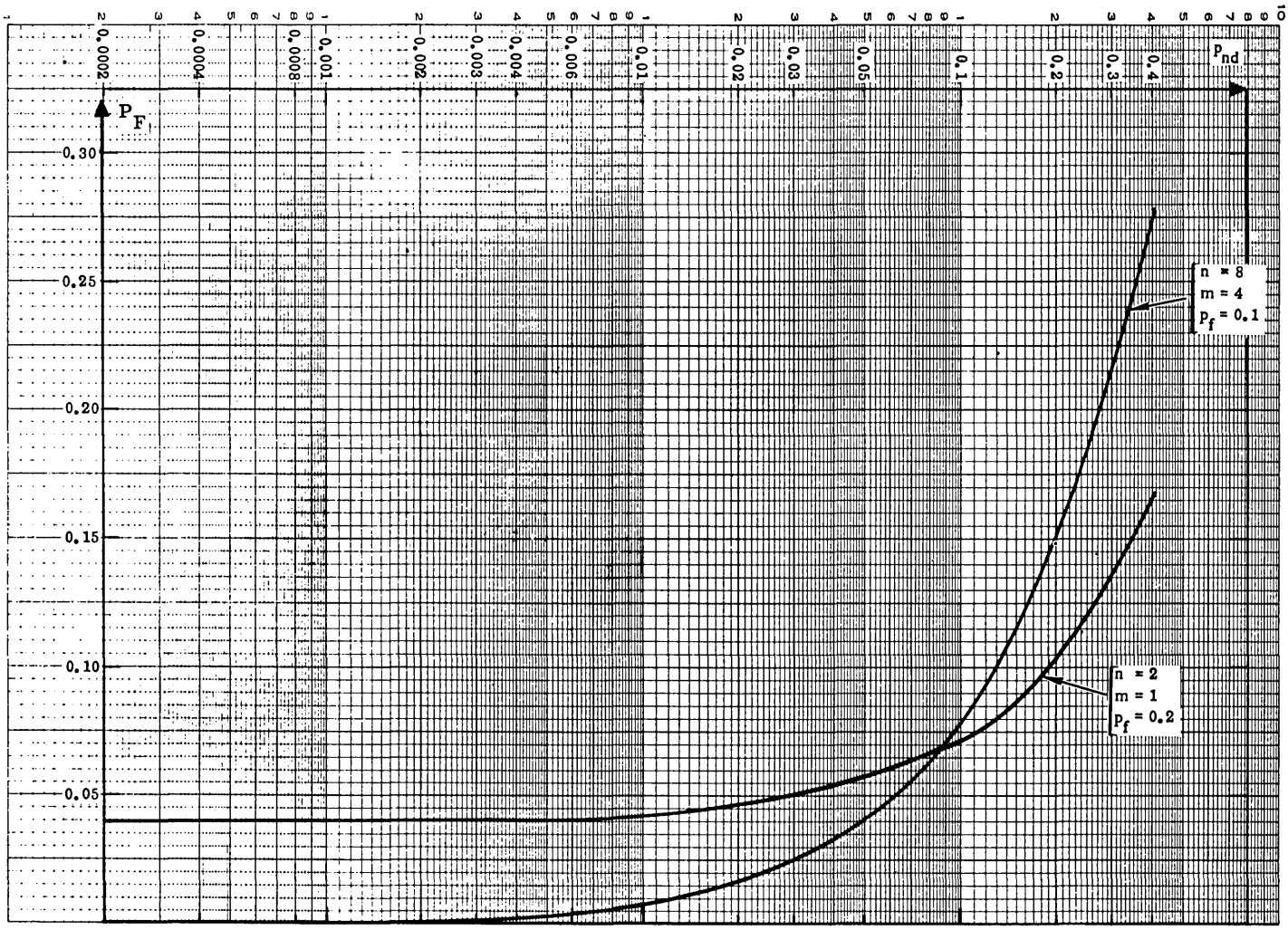


Figure 5a—Eight small modules vs two big modules: Relatively high module p_f. Quadrupling module capability is assumed to double module complexity

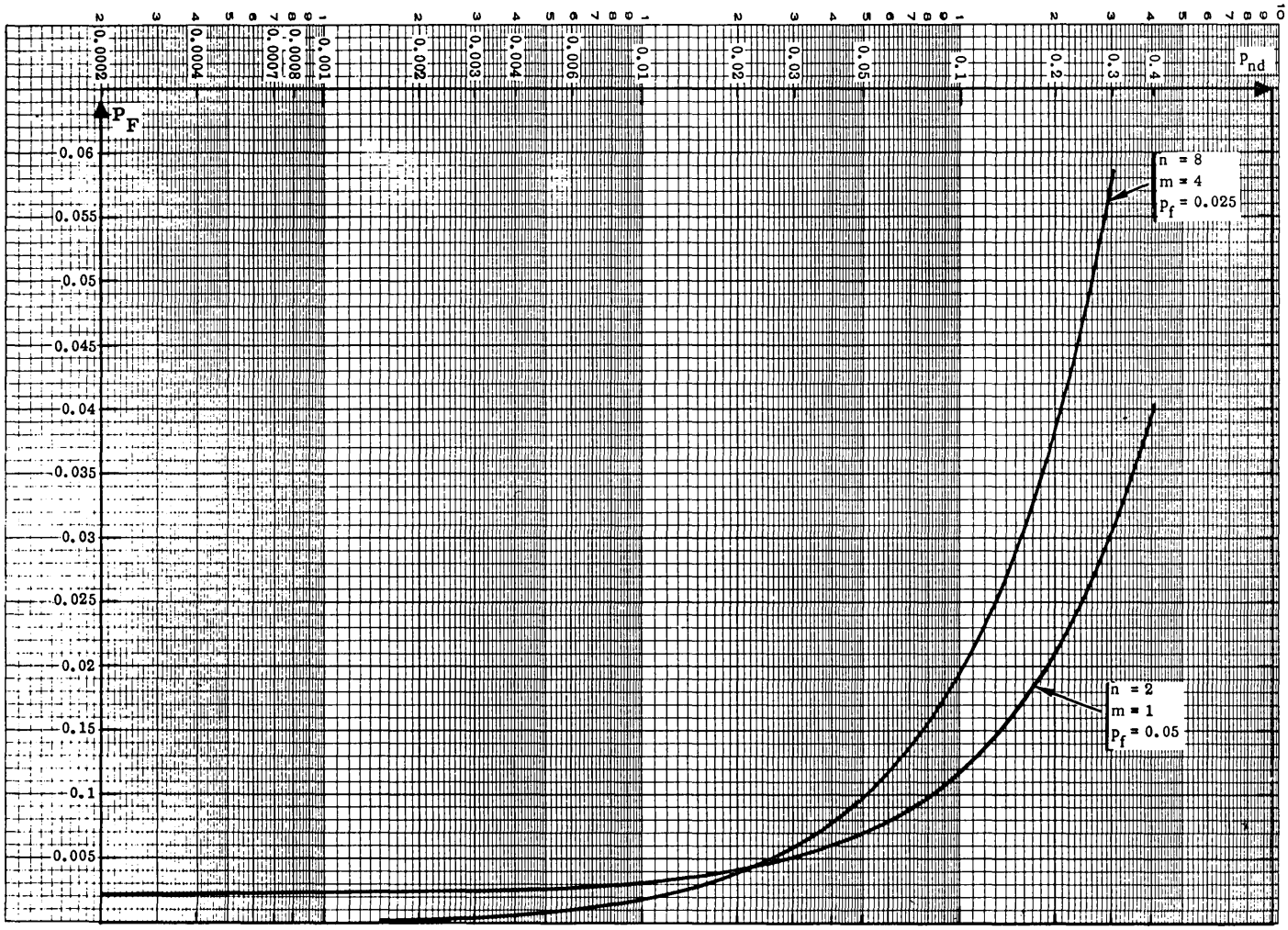


Figure 5b—Eight small modules vs two big modules: Relatively low module p_f. Quadrupling module capability is assumed to double module complexity

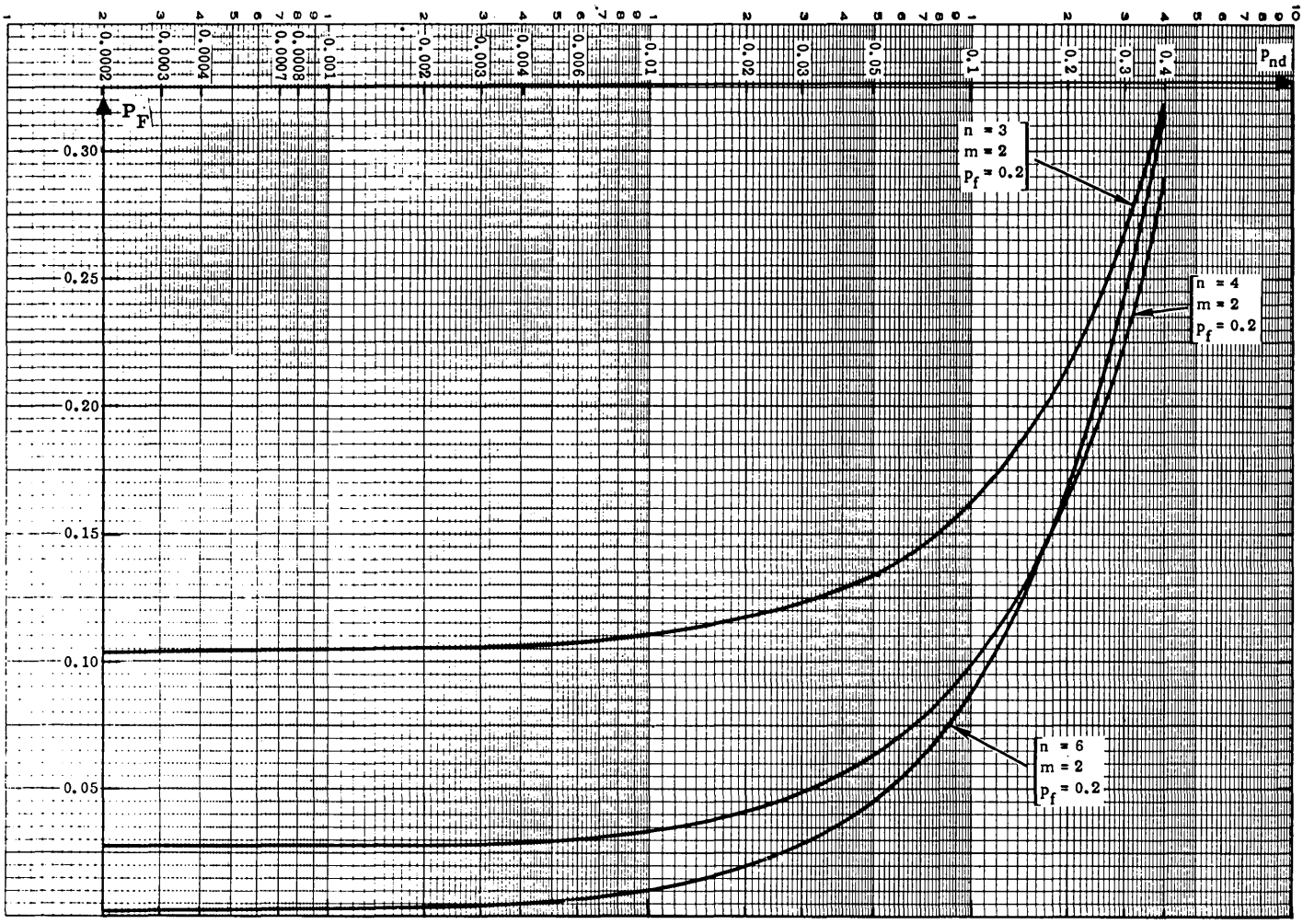


Figure 6 – How much redundancy? Module p_f of 0.2

$p_{nd:m}$ = probability of not detecting an occurred failure in a memory module

$p_{f:n}$ = probability that a processor module will fail sometime during the mission

$p_{nd:n}$ = probability of not detecting an occurred failure in a processor module

$p_{f:q}$ = probability that an intercommunication and I/O control module will fail sometime during the mission

$p_{nd:q}$ = probability of not detecting an occurred failure in an intercommunication and I/O control module

PS:m = probability of concluding the mission with at least the minimum number, (m_o), of memory modules operative and with no memory modules having undetectably failed

PS:n = similar probability for processor modules

PS:q = similar probability for intercommunication and I/O control modules.

The probability of system failure, P_F, is then given by

$$P_F = 1 - (PS{:}m)\,(PS{:}n)\,(PS{:}q) \qquad (2$$

where

$$PS{:}m = 1 - \sum_{k=(m-m_o+1)}^{k=m} \frac{m!}{(m-k)!\,k!}\,(p_{f:m})^k(1-p_{f:m})^{m-k}$$

$$- \sum_{k=1}^{k=(m-m_o)} \frac{m!}{(m-k)!k!}\,(p_{f:m})^k\,(1-p_{f:m})^{m-k}$$
$$[1-(1-p_{nd:m})^k] \qquad (3)$$

and PS:n, PS:q are given by analogous equations. Using these equations, curves similar to those in Figures 1 through 7 may be drawn for the broader

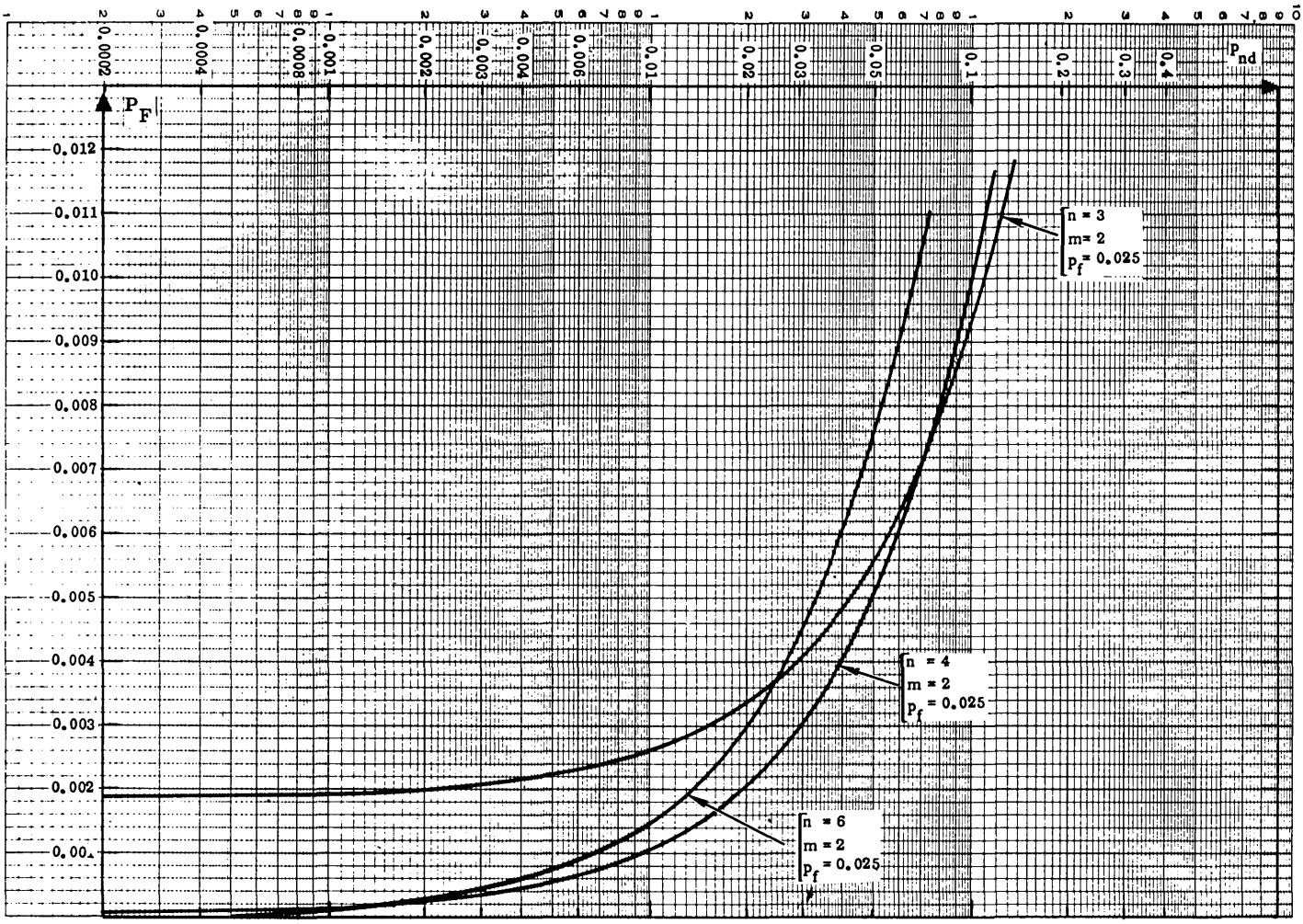


Figure 7 – How much redundancy? Module p_f of 0.025

class of reconfigurable multi-processors which contain several different module types.

E. Application to single computer with spares

The foregoing material has been derived under the basic assumption that the redundant modules of the system started out "on line"; i.e., plugged in, powered up, and possibly doing some useful computations while waiting to be automatically switched into the role of a failed, system-essential module.

If the system's redundant modules are assumed to be "off-line" (not powered up, and perhaps not even plugged in), the model is seen to be directly applicable to two specific kinds of situations and the nature of the model undergoes some change.

The first application of the model would appear to be to a spaceborne multi-processor where the spare modules might be plugged in but would not be powered up, primarily to extend their lifetime, and secondarily, to conserve power. For this case either an identical module model or an asymmetrical configuration model would be appropriate, depending upon the design of the system in question. The second application would appear to be a conventional, single computer which may be dependent upon a field maintenance facility with board replacement but not board repair capability, and with a limited number of spare boards. The number of different board types would correspond to the number of different module types in the asymmetric model of Section D. For the sake of specificity, it may be desirous to assume in the single computer application that the field maintenance facility has no elaborate check-out equipment, so that when a failed module has been replaced by a spare, the

probability of detecting a previously-occurred malfunction in the newly placed module is simply equal to the probability of detecting a malfunction which occurs in one of the on-line modules while in use.

Consider now the basic departure in the "off-line spares" model covering the above two applications from the on-line models presented in Sections B and D. It will be recalled that the system failure criteria pursued for the model of Section B included the undetected failure of any module, whether it was performing system-essential calculations or not. This criterion must now be replaced with its alternate, as defined in Section B: only undetected failures in system-essential modules contribute to system failure probability. Obviously, if a spare module not in use has an undetectable failure, it can hardly cause the system to fail until it may be erroneously used to replace a failed, on-line module whose failure was detected. This alternate failure criterion forces two changes in the model for the off-line spares system:

1. Each distinct module type must be broken into two groups. One group consists of the m modules required to be on-line for system survival. With each of the modules in this group there is associated a p_f characteristic of this module type when it is powered up. The other group starts out with the $(n - m)$ modules which are off-line spares. Its modules have another p_f, determined from an MTBF characteristic of this module type when it is not powered up and from a "mission" duration equal to the aggregate MTBF of the m on-line modules of that type.
2. Whereas the basic equation giving system failure probability, p_f, in Sections B and D did not concern itself with the order in which successive failures occurred and could thus be a simple combinatorial equation, the equation for system failure in the new model must concern itself with the order of successive failures. This can be illustrated by an example in which it is assumed that two modules of a particular type fail during a mission. One is from the on-line group, and it fails detectably. The other module is from the off-line group, and it fails undetectably. If the on-line module fails first, a good spare module will be transferred from the off-line group to the on-line group, with no resulting chance of system failure. This is followed by an off-line module failing undetectably, but since no further transfer of off-line modules to the on-line group will occur, there is no possibility of the off-line module's undetected failure contributing to system failure probability. On the other hand, if it is the off-line module which (undetectably) fails first, then when the on-line module fails there is a possibility that the failed off-line module will be chosen to replace it. This would cause system failure, and so this ordering of the two events must have a contribution to system failure probability.

   The result of the need to consider the time-ordering of these events is that the formula for P_F becomes more complex, and results in a step-by-step calculation of the contribution to P_F of each separate permutation of module failures. The results of this computation are not yet complete in the current study, but since the formulae are all straightforward, it was believed worthwhile to present here the basic approach, leaving the rest to the reader and, hopefully, to his computer. He will need one.

ACKNOWLEDGMENT

The authors wish to acknowledge the assistance of Mr. J. Hirsch, of the Autonetics Division of North American Aviation, Inc., whose instinct for the intricacies of probability formulations and whose knowledge of the self-test problem frequently prevented the authors from making gross blunders in their thinking and in their calculations.

A distributed processing system for general purpose computing

by G. J. BURNETT, L. J. KOCZELA, and R. A. HOKOM
Autonetics
Anaheim, California

INTRODUCTION

This paper presents a conceptual design of a distributed processing system aimed at providing general purpose computing and very high tolerances of failures while taking advantage of future large scale integration techniques. In addition it was desirable for the system to be easily expandable or contractable so that it could be flexibly applied to a variety of applications. The system was designed for a spaceborne application; however the structure should have general applicability to a variety of systems. A method of analyzing the parallelism inherent in the computations to be carried out on the system is also presented. The distributed processor can thus be efficiently organized for the composition of general purpose (or special purpose) computations for a given application. Thought has also been directed toward the design of a system executive program and failure detection and reconfiguration methods; however, these features are still in development and only a brief discussion of the system executive is given in this paper.

Continuing development of high density metal oxide semiconductor (MOS) circuits and bipolar arrays for use in near term and future computation systems will make the logic, or processor, sections of computers increasingly simple in comparison to the memories. This development is clarified by the fact that semiconductor memories are beginning to be investigated for the main memory in future computation systems. These memories would use many semiconductor chips, whereas a processor would only use one or two such chips. As the processing of semiconductor chips matures, the reliability of the one- or two-chip processors will also be significantly greater than that of the magnetic or semiconductor memory, except for the problems of connections and checking out the processors. It is thus clear that there is a need for new computer organizations which take advantage of large scale integration (LSI) technology in order to offer enhanced computation system features. One such organization is the distributed processor discussed here. This structure can simply be considered to be a semiconductor memory that has a small amount of each wafer (less than 1/10) devoted to processing. The system therefore consists of an array of cells where each cell integrates processing and memory onto one wafer. Such a structure then uses a small hardware increase over a conventional uniprocessor in order to offer increased reliability, flexibility, and execution speed for certain types of computations. The higher reliability is achieved for a subset of the system's tasks through graceful degradation, i.e., as cells fail, a smaller number of computations including a basic subset are carried out. The increased flexibility comes about due to the ability of the system to expand and contract in small increments to meet the requirements. The increased execution speeds can be achieved for certain sets of the computations that are amenable to parallel processing. These points will become clearer as the system is discussed in the following sections.

Natural and applied parallelism

A discussion of parallelism is given before presenting the system structure. By defining natural and applied parallelism, and by investigating the implementation of both types of parallelism in a distributed logic structure, a perspective on the use of local and global control may be gained. Local control simply implies control of a cell by its own control unit; global control implies common control of a number of cells by a central control unit.

Natural and applied parallelism may be defined as follows:

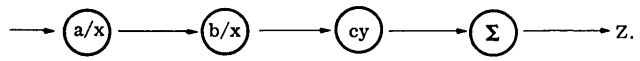
1. Natural parallelism: A property of a set of computations that enables a number of groups of

operations within the set to be processed *simultaneously* and *independently* on distinct data bases or on the same data base.

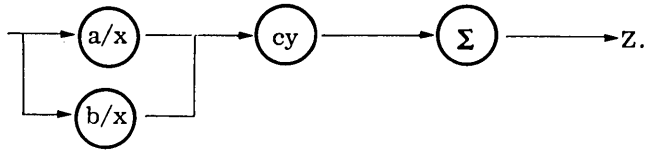
2. Applied parallelism: A property of a set of computations that enables a number of groups of *identical* operations within the set to be processed *simultaneously* on distinct data bases or on the same data base.

It may be noted from these definitions that applied parallelism is actually a special case of natural parallelism, since the naturally parallel operations could be groups of identical operations. However, the distinction is made since it has important implications with regard to computer organization, as will be pointed out later in this section. The following simple example will serve to illustrate these definitions.

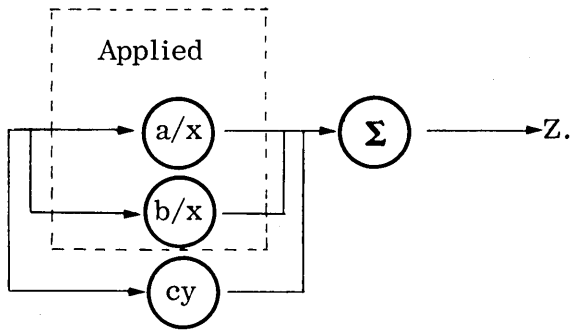
Consider the calculation of Z from the equation, $a/x + b/x + cy = Z$. This could be calculated on a single processing center, as follows:



However, taking advantage of applied parallelism in an appropriate processing system would give the following calculation:



If natural parallelism could also be taken advantage of in a processing system, the computation could be executed as follows:



It should be noted that although this example demonstrates the application of applied and natural parallelism to a computation, larger groups (larger than simple parallel divide or multiply operations) of operations such as a number of parallel calculations on separate data of a matrix multiply routine (applied parallelism) are generally considered. It should also be noted that the natural parallelism to be considered here is not only within a particular task but also between separate tasks, such as between two users of a ground system or between the navigation programs and the status monitoring programs in a space system.

From the foregoing it may be seen that computations containing applied parallelism may be efficiently implemented on a distributed processing system such as the Solomon machine (see Reference 1), which uses a central control unit and a number of cells to carry out the instructions from the control unit. This type of global control (central control unit) can efficiently handle applied parallelism since the central control unit provides common instruction storage as well as some common data storage. The common control unit also saves some control hardware; however if future systems are constructed from LSI technology, this savings will not be as great, since logic will be inexpensive and since a centralized control unit requires more connections and decoding hardware than local control. The significant disadvantage of a global control structure for general purpose computations is that it cannot efficiently execute naturally parallel computations. Such computations must be executed in a sequential manner, with the central control unit using only one cell at a time. As a result, even though a global control structure saves storage over a local control structure when applied parallelism is available, the hardware utilization of a global control structure is relatively low on most sets of general purpose computations.* Computations containing natural parallelism may be efficiently executed on a distributed processing system, such as the Holland machine (Reference 2), which has a number of cells, each with some measure of local control. The independent parts of a computation may then be executed in separate groups of cells. Theoretically, a local control structure could achieve high hardware utilization if many naturally parallel computations were available. However, in a large system with spacially oriented communications (neighbor to neighbor communications rather than common bus) like the Holland machine, the programs typically must be optimally placed in order to limit communications. This placement of programs makes programming very difficult. It is significant that a local control structure could certainly carry out both applied parallel and naturally parallel computations;

*One method of improving this situation presently is being implemented in ILLIAC IV, which uses a number of groups of cells, each with its own central control unit; however the hardware utilization within each group would still be relatively low on naturally parallel computations.

but execution of the former requires repetitive instruction and data storage, along with repetitive control generation. Thus, execution of applied parallel computations on a local control structure results in inefficient use of the storage available to each cell (or in each cell).

Local control structures are most efficient when executing naturally parallel computations, whereas global control structures are most efficient when executing applied parallel computations. Consequently, if both control features could be efficiently combined into one structure, a very powerful distributed processing structure could result. Such a structure is described in the next section.

Distributed processor structure

A conceptual description of a new distributed processing structure capable of carrying out general purpose computations and taking advantage of both natural and applied parallelism is presented. The machine is composed of cells capable of operating under local or global control. This control should enable the structure to obtain efficient hardware utilization while reducing instruction and data storage. Significantly, this structure was not designed, as distributed logic structures have been in the past, to solve a particular type of computational problem. Instead, it was designed as a general purpose computer to take advantage of the new LSI (including MOS/Silicon-On-Sapphire) technologies and to provide the other features mentioned in the introduction.

The following description of the distributed processor gives a comprehensive conceptual description of the machine. Further developments of the explicit features of the machine are presently being studied and will depend upon trial programming effort and software development. As a result of the uniqueness of this design and of the lack of programming experience with distributed logic structures in general, features such as the length of the instruction word or a listing of the instructions cannot be given at this point.

General organization

The distributed processor structure uses a number of fixed size cell groups, each carrying out a particular task or tasks, to handle the system computation load (see Figure 1). The bulk memory (also shown) is used only for loading programs and data since the cells themselves provide all the main system memory. The conditioners and sensors are shown to represent the peripheral equipment and controllers connected to the computation system. Each cell is planned to be

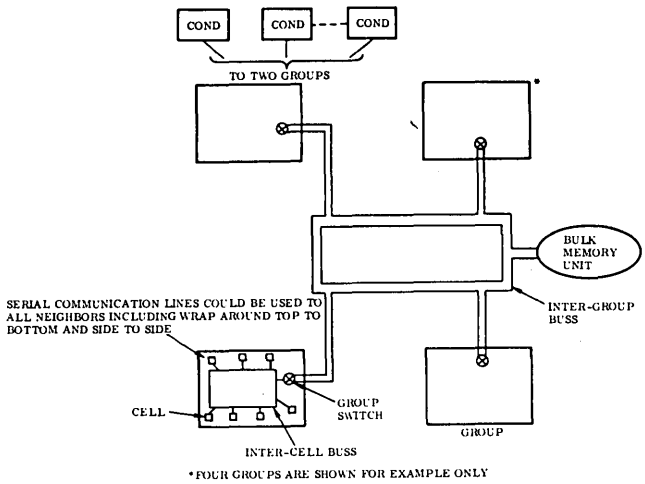


Figure 1 – Distributed processor system

a single MOS (or MOS/SOS) wafer although initial prototype and near-term systems would be constructed using more than one chip to make up a cell. (The one-wafer-per-cell design would offer the highest reliability and the advantages of only one chip type, but it will probably not be practical until the middle to late 1970's.) MOS technology has been selected since it will offer very high circuit densities in the near future, along with low power requirements. (These attributes are especially desirable in space and avionics applications.) The cell will contain 512 eighteen-bit registers for memory and control storage, while the remainder of the cell will be used for a processor and line drivers. The processors will be capable of operating with at least a few-megahertz clock. The cell operation will typically be communication limited, so that the actual operation of the cell will be slower than this. Slower operation will then loosen the constraints on the memory section of the chip and lower power dissipation. The cell structure will be described later in this section.

A group of cells may carry out a complete task, a number of small tasks, to even part of a large function or program. One of the primary considerations used in dividing programs among the groups is to limit the intercommunication between groups as much as possible. In this way the inter-group buss can be used primarily for communication of I/O variables and for communication to and from the groups operating as the executive. (Note that the executive program will simply be loaded into any group. There will be no need for special features.)

A second consideration in dividing programs among groups is to enable as much global control to be used as possible in order to take advantage of applied

parallelism, and thus save storage. Within a group, any cell will operate as a controller and provide commands for the others. The other cells can either accept these commands or execute commands from their own memories. In this way, both local and global control may be carried out simultaneously within a group. At the same time, any number of groups may operate in parallel. This type of operation enables both the natural and applied parallelism inherent in a system's tasks to be efficiently carried out.

The size and number of groups necessary for a given application are dependent on the parallelism associated with the requirements. As a result this topic is discussed in the next section. However an example can be given here of a system roughly optimized for a central general purpose computer in a spaceborne Mars Lander Mission: it requires four groups of twenty cells per group and provides approximately 40,000 words of instruction, control, and data storage at any point in time during a mission. (The bulk storage unit loads in new information for each mission phase.)

Some additional features of the distributed processor not shown in Figure 1 are necessary if the system is to remain capable of continuous operation after a number of failures. A second inter-group buss, connected by a separate group switch to each group, should be included. This, of course, allows for continued computation after a buss failure. Spare cells may also be maintained on-line within each group in the system, thus increasing system availability. This type of sparing philosophy is practical here since adding extra cells simply involves extra connections to the inter-cell busses.

System intercommunication is depicted in Figure 1. Each cell may communicate with its four neighbors serially and in a byte parallel manner over the inter-cell buss to the other cells in its group. The communication rate and the number of bits in a byte on the inter-cell buss must be specified for the required system; however, one- to two-megacycle communication rates should be practical.

Storage in the distributed processing system is volatile semiconductor memory. No problems should occur as long as power is continuously applied. The primary power supply would be backed up and could be fed into the distributed processor on separate, isolated lines. These lines could then be connected to the computation system through separate voltage regulators with diode isolation. If one of the supplies were to go out of tolerance the other would simply take over. In this manner, the power supply could be made more reliable than the computation system hardware. Another simple possibility would be to use presently available small nickle-cadmium batteries to provide backup power for the computation system. This would be practical since a system such as the one mentioned above (four groups of twenty cells per group) could be constructed from MOS/SOS chips and would dissipate only a few watts of power.

Advantages and problems

Before presenting additional features of the distributed processor, some discussion of the advantages and problems of the structure is in order. The primary advantages have been pointed out to be high reliability, applicability with good hardware utilization to general purpose computing, good use of future LSI technology, and easy expandability or contractibility. The principal problem associated with the structure is making it easy to program. These points are clarified below.

The ability of the system to offer high reliability can be interpreted as the ability to continue executing a critical or high priority subset of the computations in spite of hardware failures. In other words the computations will be "gracefully degraded." This is possible since the system is divided into a number of independent modules (cells) that can fail without bringing down other parts of the computing system. In fact a few spares will be kept on line in each group so that a few cells can fail and simply be replaced by the spares without affecting computation. However, after these first initial failures, the computation power of a group would simply degrade in small increments with each additional cell failure. It should be noted that to properly take advantage of this graceful degradation ability a high probability of failure detection must be achieved for each cell, and the number of single system failure points must be minimized or eliminated.

Another way to look at the graceful degradation ability of the system is to say that it is designed to be tolerant of cell failures. It is tolerant in the sense that not only can a number of cells fail and simply be replaced by spares connected into a group, but also cell failures beyond the number of spares available only cause small degradations of the system. A little reflection will show that this type of operation is significant since actual on-line failures caused by overstresses, for example, can be tolerated along with a reasonable number of on-line failures caused by use of originally defective parts. This failure distinction is a fine point, but it is important because it has been observed that a great many failures in complex IC's are actually caused by manufacturing defects that were not detected prior to placing the system on-line. In fact, many of these failures only occur under a very specific set of circumstances and so may not

cause on-line failures for thousands of hours. These failure modes can generally be removed by feeding back information to the production line, but it is a long and expensive process to obtain very reliable components. Of even more importance is the fact that as circuitry reaches greater and greater complexity on a single chip, the practicality of giving a chip a 100 percent test on the ground before inclusion in a system decreases immensely. As a result, in the future the ability of the computation system to tolerate a number of latent manufacturing defects will gain more significance. It should also be noted that this ability to tolerate cell failures does not exist to the same extent in a global structure such as the Solomon machine, where a single failure in the main control unit or memory could bring the whole system down.

The distributed processor is capable of efficiently executing general purpose computations by taking advantage of both local and global control as described below. In addition, each cell has a reasonably large instruction set and memory. Because this large capacity significantly eases programming and communication problems typically encountered with distributed processing organizations, relatively large subtasks can be executed by a cell.

The use of local control enables each cell to execute computations simultaneously with other cells as long as the computations in execution at one time can be carried out independently of each other. This is simply the obvious restriction that separate cells cannot operate simultaneously on sequential sections of a single task or related tasks if the computations in one part depend on the results of another part. For tasks that are sequential in nature, this restriction means that only one cell can be used to execute the computations at any given time. As a result of the sequential restriction during certain periods the distributed processor may operate in a relatively inefficient manner due to the fact that some of the hardware will not be utilized. However the availability of separate independent tasks, such as the navigation and guidance task and the status monitoring task in a spacecraft, should minimize the occurrences of inefficient hardware utilization periods. In contrast to the above, this same local (and global) control enables the system to use its parallelism to achieve high computing speeds if the computations are somewhat specialized so that they contain a large amount of parallelism. In particular, the best use of the cells for speed is achieved when the computations contain a good amount of applied parallelism and/or small high rate programs that are naturally parallel.

The processors in the system will be fairly powerful so that at times they will require the use of more memory than the 512 words that are available within their cell memories. This can be accomplished in three ways. One method allows one cell to use another cell's memory via the communication busses. This enables the cells with small high-rate programs to share their memories with other cells that are memory restricted. A second method of alleviating memory limitations is providing the system with the ability to use global control where it is applicable. (As mentioned earlier, global control is characterized by the use of a single instruction stream on multiple data streams and is therefore useful in problems that require the same set of operations on different pieces of data, e.g., matrix manipulations, etc.) The distributed processor takes advantage of global control by using one cell to hold the instruction stream and send the stream over the communication buss to the other cells in a group executing the same problem. In this manner, redundant storage or sequential redundant access of the same program is avoided. The use of global control as discussed here enables a number of processors to be used to execute certain computations with only a minimal amount of memory. A third method of decreasing the amount of storage required per cell is to provide less buffering in the memories for I/O data. This would increase the required I/O rates, but that is acceptable since extra processing is available. In addition, one of the groups operating as the central executive will provide a storage area for common variables and constants if appropriate.

The distributed processor takes good advantage of future LSI technology in a number of ways. The cell has a fairly regular structure so that it can be fabricated with extensive use of discretionary wiring or similar techniques that enable the bypassing of bad elements to increase yields. For example a number of extra memory cells can be fabricated on the 9/10's of the wafer (or wafers in near term systems) devoted to memory so that discretionary wiring can be used to connect up enough good cells for 512 eighteen-bit words. Correspondingly, more than one processor can be fabricated on its section of the wafer so that processor yields can also be increased by discretionary wiring. The cells are also functionally complete on a single wafer so that each wafer requires only a relatively small number of external connections. (This will of course not be true in near term systems.) Another useful feature is that the cells are used in a fixed connection pattern that can easily be placed on a board or other appropriate package.

Expandability is easily and flexibly provided since system size can be varied in small or large increments by simply adding or taking away the required

number of cells. This expandability may find even more importance in providing the system the ability to turn cells on and off as required during a long space mission or during various periods of computation in a ground-based system. This mode of operation will substantially decrease the power dissipation of the system and may also increase the reliability (assuming off-line or dormant equipment is more reliable than on-line or operating equipment).

The main anticipated difficulty with the distributed processor is in making it relatively easy to program. The tasks must be split into subtasks that, when programmed, will fit into a single cell. This splitting should also give as much freedom as possible in scheduling the tasks on cells; that is, constraints on executing certain computations in a sequential fashion should exist, if possible, only within the subtasks to be executed on an individual cell. The allotted subtasks must then be assigned to the minimum number of cells. In addition to the assignment, the subtasks must be scheduled so that all of them can be accomplished in the required period of time. This requires consideration of allocation of separate (non-overlapping) periods to the global control programs and to avoiding conflicts on the intercommunication busses. Clearly, the scheduling operation could also feedback on the assignment of subtasks to cells such that a new assignment is necessary. These problems are discussed in more depth later.

From the foregoing discussions it can be seen that actually programming a subtask for a cell would not be much more difficult than programming the same subtask for a single computer system; however, if the programmer is required to subdivide tasks, assign them to cells, and set up the scheduling, the programming job will be relatively difficult. Therefore, one area to be investigated in the future involves the development of software packages to automatically subdivide programs into tasks that can be executed by a single cell, to hand'e the automatic assignment of subtasks to cells, and to schedule automatically the global control programs and intercommunication on the busses. These same software packages will be useful for reconfiguring the system after failures.

Explicit features

The explicit features and operation of the system are still under development and will be influenced by future programming and analyses. However, the features mentioned earlier, plus the following description, will certainly be representative of the final system.

The cell structure is shown in Figure 2. Each cell operates as the controller of a group or part of a

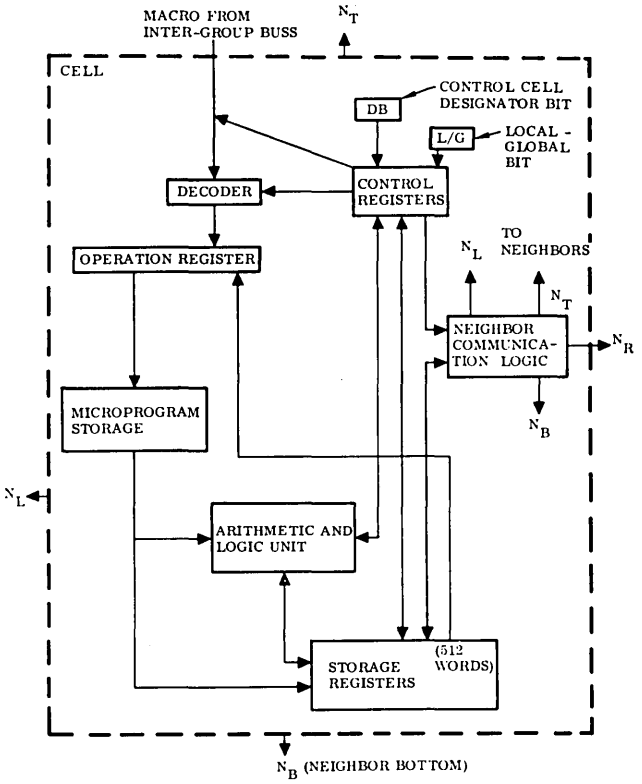


Figure 2 – Distributed processor cell

group, an operating cell, or a storage cell. The controller cell specified by the designator bit being set to 1, can provide global control for a task by placing macro instructions on the inter-cell buss and by controlling the use of this buss by each cell. The macro instructions would be commands such as vector dot product, check sum, sine, add, etc. The operating cells receive their macros from the controller cell, if they are used in global control, or, if they are under local control, from their own storage registers. The macros are then decoded and used to read out a sequence of operations from the micro program storage contained in a cell (see Figure 2). The sequence of instructions from the micro program storage causes storage registers and control registers to be added, exchanged, transferred to neighbors, etc.; thus the control registers allow a macro from the inter-cell buss to load the operation register (global control-L/G = 1) or a macro from a cell storage register to load the operation register (local control-L/G = 0). The next operation is obtained by sequencing to the next instruction in a cell's internal storage or in the controller cell's storage.

For some computations, a cell may need more storage than is available within itself. It can then use one of its neighbors or any other available cell

in its group as a storage cell. The inclusion of the neighbor communication logic and connections will depend on future investigation. The neighbor communications will only be included if a definite need is determined since they give the system a spacial dependence. This dependence increases the difficulty in assigning programs and of reconfiguring the system after failures. If included, the neighbor communication logic is used to obtain, from a cell's neighbor, information necessary to the computation being carried out. This information could, of course, be either additional commands or data. Note that certain commands on the inter-cell buss will unconditionally set L/G to 1 so that the central executive (located in one group) can take over the operation of any cell. Cells can then be reassigned to new tasks by loading from the bulk storage unit.

All storage within a cell is directly addressable and is divided into control registers and storage registers. The storage registers hold the programs and data to be used by this cell and possibly other cells. The control registers are the general processor registers used in the execution of instructions. The controller cell controls the group switch shown in Figure 1. The group switch contains a small amount of decoding logic and a flip-flop register. If the inter-group buss contains a command for this group, it will be recognized, accepted, and let on the buss to be received by the controller cell. This group will then remain connected to the inter-group buss until the transmission is completed. If the command is not for this group, the group will not be connected to the inter-group buss. (The command from the executive group contains a task name that can be recognized to belong to a certain group.)

The executive can be located in any one or more of the groups. It is responsible for the following tasks: (1) controlling the inter-group buss, (2) handling I/O, (3) handling communication with the bulk storage unit, (4) holding global data so that any group may use it, (5) handling data communication from group to group, (6) sending out macros to load the system, and (7) allocating I/O time on the inter-group buss. The operation of the executive is briefly described later in this paper.

Information sent over the inter-group buss is byte parallel. The number of bits in a byte is determined from the number of groups present, the speed of the cell logic, and the required system computation speed. However, for any particular system, making the buss less parallel means that less drive is required of the cell, resulting in lower power. Higher reliability is also obtained due to fewer connections and drivers. The groups can use the intergroup buss only when sampled and allocated time by the executive groups. Information on this buss is tagged with control bits, names of tasks, data addresses, and data itself.

The foregoing description in this section specified the distributed processor structure, its useful features, and its problems. The next section discusses the methods of analyzing parallelism in computations so that the structure can be put to good use.

Evaluation and application of parallelism

In order to take maximum advantage of the distributed logic structure the computations for a given system must be investigated for both natural and applied parallelism. These investigations are necessary under the following two circumstances. If the computer is to be constructed for a particular application, parallelism investigations will have a large influence on choosing the following parameters: cell speed, amount of storage per cell, number of cells in a group, and number of groups. This investigation is discussed in the majority of this section. Once a structure is selected, or, if the computations are to be carried out on an existing distributed processor, parallelism investigations would be used to aid in distributing tasks throughout the computer, as discussed in the last paragraph of this section. In order to carry out the parallelism investigation, flow charts are plotted for the various functions of the set of required system computations and these are analyzed for parallelism. The flow charts are then updated as the parallelism is exploited so that the branches within a function that can be carried out in parallel are shown. These diagrams will also take into account the separate functions that can be carried out in parallel (natural parallelism). Examples of representative flow charts are shown in Figures 3 to 5.

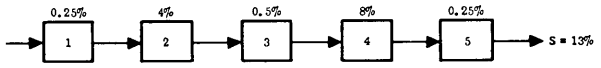


Figure 3 – Sequential computational execution

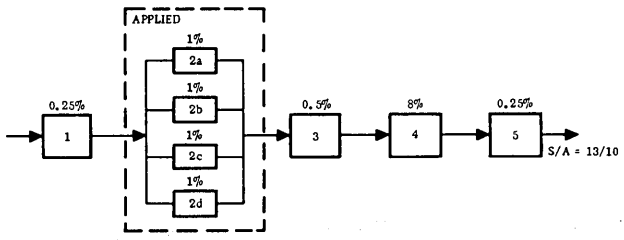


Figure 4 – Flow diagram utilizing applied parallelism

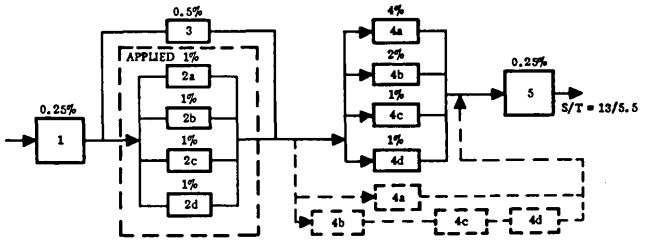


Figure 5 – Flow diagram utilizing applied and natural parallelism

General considerations and flow charting

The theoretical approach to the parallelism investigations necessary to give a precise organization involves constructing a number of curves from the flow charts and then using these curves to obtain relatively optimal values for the organization parameters. Unfortunately, this approach, if carried out in detail, is very complex and involves a considerable amount of feedback to update the curves. It is much too complex to be practical. Simplifying assumptions can be made so that a less complex approach can be carried out to obtain approximate values for the parameters. Such an approach is described below; a detailed discussion of the complete investigation is given in Reference 4.

An important consideration in the analysis of parallelism is the degree of parallelism utilized and the reduction in computation time or the effectiveness of the parallelism. In order to obtain a quantitative evaluation for the decrease in execution time from exploitation of parallelism, two ratios are calculated from the flow charts of the computations. The first ratio, S/A, represents the time to execute a function on the distributed processor utilizing applied parallelism divided into the time to execute the same function in a sequential manner. The second ratio, S/T, represents the time to execute a function on the distributed processor utilizing both types of parallelism divided into the time to execute the same function in a sequential manner. These ratios are termed the computation reduction ratios. The ratio S/T can be correlated to the speed of a cell by realizing that for a given task a reduction ratio of X means that a cell would only have to operate at 1/X the speed of a single processor capable of executing the same task. Since system requirements are typically specified in terms of required operations per second on a uniprocessor, the T/S reduction ratio for all the system tasks, with any specified degree of parallelism, can be directly and simply converted to required cell speed. All that is necessary is to multiply S/T times the total requirement.

When exploiting parallelism within system flow charts, the aim is to use the minimum degree of parallelism necessary to obtain large enough computation time reduction ratios to be able to execute all tasks in the required periods of time. The above point is made clear by considering that implementation of parallelism in the distributed processor system required *at least* one separate cell for each degree of parallelism (for each subfunction to be executed in parallel with other subfunctions). (More than one cell could be required, depending on the amount of storage necessary to implement a given subfunction.) For example, it would not be worthwhile to execute all possible computations in a naturally parallel manner if the results of these computations could not be used until another longer computation was completed. Only enough computations should use natural parallelism so that all the calculations end at approximately the same time. The remaining calculations simply time-share the number of cells necessary to execute them sequentially. In this way, the maximum speed increase is obtained, while using the minimum number of cells. The above discussion, then, suggests that the best way to schedule computations is to start the longest ones first on as many cells as are available. Whenever a cell becomes free, the longest computation left is started.

In addition to the above mentioned uses of parallelism it should be clear that as much applied parallelsim as practical should be first exploited when setting up execution of a set of tasks. This is the case, since use of applied parallelism saves control and constant storage over a naturally parallel execution of the same tasks. In fact, in order to enable the applied parallelism to be adequately exploited the distributed processing system was divided into groups. If the system was not grouped all the applied tasks would have to be executed sequentially since each must use the central buss and controller cell to communicate to cells in the global mode. This restriction would certainly hamper the scheduling of programs and the use of parallelism. As a result the grouping of cells in the machine enables a number of applied tasks to be simultaneously executed, if desired, without considering extensive sequencing.

Following is an example which should clarify the above points. A hypothetical computation shown in Figure 3 depicts a sequential execution of the various calculations, Block 1 first, etc. The percentage over the blocks gives approximate percentiles of computation time used to execute a given block (purely sequential execution within all blocks also). As discussed earlier, the first step in exploiting the parallelism in the computation is to search for all

places where applied parallelism can be used. This could give a flow diagram as shown in Figure 4. In this diagram 2a, 2b, 2c, 2d, could by any calculations on separate pieces of data that can use a single instruction stream. Simple examples are trigonometric functions or matrix multiples. Whatever the function, each block of 2 carries out exactly the same operation. The applied ratio S to A is 13/10. The ratio is obtained with a maximum degree of parallelism of four (four subfunctions executed in an applied parallel manner). The degree of parallelism is not directly convertible to number of cells since a subfunction may require a number of cells to store all the information necessary for computation. However, four cells (corresponding to the degree of parallelism) would be the minimum number of cells that could be used to take advantage of applied parallelism of degree four. Figure 5 shows the same flow diagram, now adjusted to take advantage of all parallelism. The total ratio S to T is 13/5.5. This ratio is obtained with a maximum degree of parallelism of five (two and three in parallel). Notice that the use of natural parallelism has enabled Computation 3 to be done in parallel with 2. This, of course, saves only 0.5 percent computation time since the calculations cannot go on to 4 until 2 is completed (1 percent computation time). The subdivision of computation 4 into four naturally parallel tasks demonstrates a statement mentioned earlier: use the minimum degree of natural parallelism that will give the maximum speed increase. All subtasks of 4 must be completed before 5 can begin execution, as a result using separate cells to compute 4a, 4b, 4c, and 4d in parallel is unnecessary since 4a takes longer than all the other tasks. 4b, 4c, and 4d would be computed sequentially (shown in dotted lines) on as few cells as possible since the execution time would then be 4 percent which is of course, the same execution time that would result if all subtasks were processed in parallel.

It should also be noted that Computation 2 could be done in naturally parallel fashion, but it is shown in the figures as an applied calculation since this provides more operating efficiency. The figures could also have shown nesting of parallelism within parallel blocks; however, this generally occurs on a larger scale. For example, other separate functions could be executed in parallel with Computations 1 to 5. This would give a flow diagram for the whole system that had a number of branches like that shown in Figure 5 being executed in parallel. These branches would vary in length (or execution time), but they would be organized so that they all end at approximately the same time as the longest branch, thus making the best use of the cells available.

Parallelism curve construction for system specification

Methods of analyzing computations to determine a minimum value for the reduction ratio S/T have been presented. Limiting the degree of parallelism available and determining the maximum reduction ratio with each of a number of degrees is the next consideration in determining the system organization. The procedure for obtaining the maximum computation reduction ratios is the same as that above except that the lower degrees of parallelism force more tasks or subtasks to be executed sequentially (i.e., the branches like that shown in Figure 5 are lengthened somewhat). If the degree of parallelism is varied from one to some maximum usable value and if the system flow diagram (with branches as shown in Figure 5) is permuted in order to obtain the maximum reduction ratio (S/T) for each degree chosen, a curve as shown in Figure 6 can be obtained. (This curve is developed

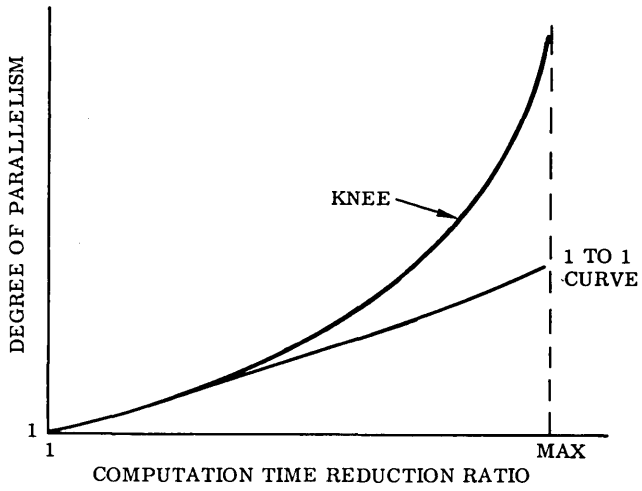


Figure 6 – Degree of parallelism vs computation reduction ratio

if all the cells are in one group and if as much storage is available as needed.) In order to choose a degree of parallelism for the system that is relatively efficient, this curve should be compared to the 1 to 1 curve. (The 1 to 1 curve has a constant slope indicating that an additional incremental increase in the degree of parallelism will give a constant incremental increase in the computation reduction ratio.) In other words, a range of parallelism should be specified on the curve at a location that is not too widely divergent from the 1 to 1 curve, i.e., the knee of the curve. The knee has a rapidly changing slope, indicating that an additional increment in degree of parallelism provides a smaller increase in computation time reduction ratio.

In order to choose an approximate explicit degree of parallelism for the system the necessary speed of

the cell must also be considered. This cell speed is specified, as mentioned earlier, by simply dividing a chosen reduction ratio on the horizontal axis into the total required operations per second for the system. Although the possible operating speeds of the cells can vary over a fairly wide range up to a few megahertz logic clock, it is desirable to limit the cell speed somewhat so that the memory electronics and operation can be simple. As a result, the desirable range of cell speeds may somewhat limit the range of degrees of parallelism obtained above from Figure 6. Before a particular degree can be chosen and converted to the number of cells required, however, consideration also must be given to the storage required per cell or per degree of parallelism. (It has been assumed that as much storage was available as required.) In order to actually determine the required storage, a curve relating degree of parallelism to storage, as shown in Figure 7, must be constructed from a programming consideration of the system flow diagrams developed earlier. In the actual development of this curve the system flow diagrams need to be rearranged so that cell storage requirements are as nearly equal as possible. Due to a new set of flow diagrams, the construction of this curve would feed back to update Curve 6.

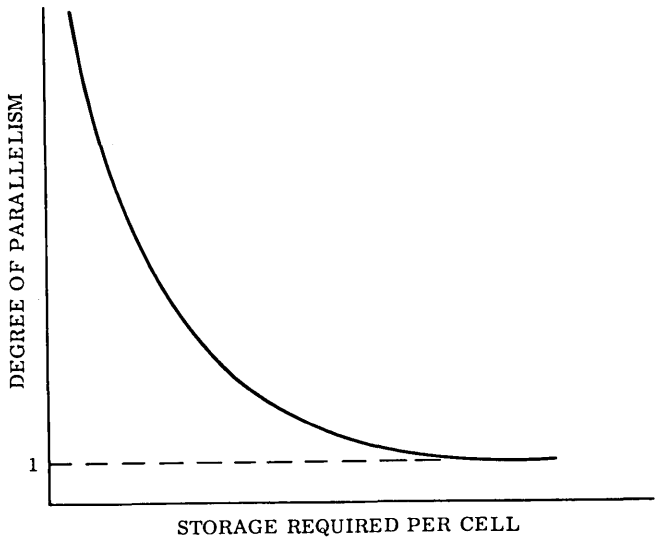


Figure 7 – Degree of parallelism vs. storage required per cell

Figure 7 is very difficult to construct accurately due to the large amount of rearrangement necessary with the flow diagrams for each degree of parallelism investigated. Because of this, the actual curve constructed would only be an approximation to the true curve. An even greater complication is introduced by the fact that the true curve will vary with the number of groups used in the system. Figure 7 is constructed assuming one group, but, as mentioned earlier, more than one group would be used so that all applied parallelism could be executed without conflicts on the buss. As the number of groups are increased the amount of storage required per cell would actually decrease due to increased use of applied parallelism. However, for an approximate or practical solution this effect will be disregarded, and once Figure 7 is obtained, assuming a single group, the range of degrees of parallelism developed from Figure 6 could be investigated for storage requirements. If the storage axis is assumed to represent the storage per cell, the degree of parallelism range from Figure 6 can be further restricted to a value with a desirable storage requirement. This degree of parallelism can then be considered to represent the number of cells required in the system.

The investigation to this point has resulted in an approximation of the speed of a cell, the storage in a cell, and the number of cells in the system. With these parameters roughly specified, the updated flow diagrams of the system can be investigated to determine roughly the number of groups required to enable the applied parallelism to be efficiently utilized. (For spaceborne applications, this number was generally small, such as 4 to 10.)

Given the choice of the number of groups, the system structure can be completely specified for the given application; but it should be realized that the approach specified above is not unique in that it could just as easily have begun with consideration of storage (Figure 7) rather than of computation reduction ratio. An answer similar to the approximate procedure would be obtained. In fact, the same answer would be obtained if an exact procedure is used (including reconstruction of Curves 6 and 7 as information on storage, the number of groups, etc., is determined and used to reconstruct the system flow diagrams).

Parallelism investigations must also be carried out in order to program a given application on an existing distributed processor or on one that has been specified for an application such as the one discussed above. This involves, essentially, assignment of programs to cells or groups and the scheduling of these programs within a group. These investigations entail laying out the system flow diagram in the manner described for Figure 6, with the degree of parallelism specified. The only differences are that the size of the groups, speed of the cells, storage in the cells, and number of groups are known. The system flow diagram is then simply permitted to obtain relatively optimal usage of the hardware for the given organizational constraints. The flow diagram is then used directly for code generation.

Software and failure considerations

Software and programming techniques for the distributed processor have not yet been studied in depth. As a result only a short synopsis of these features will be presented.

The basic programming difficulty in a distributed processing machine lies in trying to achieve optimal coding, i.e., maximum machine utilization and no usage or communication conflicts. (This is simply the assignment and scheduling of tasks on cells that was described in the last section.) These difficulties may be compounded when it is necessary to reconfigure programs after failures. One of the main programming advantages of the distributed logic structure discussed above is the fact that it is modularized into relatively small groups. This approach permits programming of the system functions into locally optimum, group-size task modules. Optimizing programs on groups of about twenty cells is much simpler than trying to optimize and reconfigure programs within a system of a hundred or more cells.

The system software packages will have to carry out such tasks as controlling communication on the inter-group buss, reconfiguring after failures, monitoring requests, supervising input/output, and carrying out self-tests. Most of these tasks are necessary in standard computation systems. The central executive sequencing of programs or tasks is minimal (it is carried out locally within a group), but a new function —inter-group communication control—is necessary to control data transfers on the inter-group buss. These transfers are due to the following:

1. Global data—those parameters of interest to more than one independent function—must be passed from the task/module which computes it to those that use it. This is not done directly, but through a global data area in the central executive.
2. Some mission functions are too large to be programmed in only one task module. When multiple task modules are used, the interface data necessary to connect the parts of the function must be passed.
3. The I/O system will use this buss to pass some data to and from task modules.
4. Executive macro instructions are issued on the inter-group buss for such tasks as loading a group from bulk storage or sending I/O data to a group.
5. Messages to the executive must be passed from the task modules.

In order to avoid conflicts on the inter-group buss the buss monitor program allots weighted usage on a time-shared basis. Each task module, including executive groups, is simply allocated a certain number of accesses per cycle. The monitor program, then, just selects, at a fixed rate, particular task modules.

The self-test programs to be carried out by the software also present some unique and interesting problems. The approaches investigated so far have considered both testing using operational data and redundant cells, and testing using self-test programs and floating cells or floating groups. These approaches are still under investigation and so will not be presented here. It is of interest that the self-test hardware and software will have to be very complete in order for the system to truly take advantage of graceful degradation, i.e., the P_d (probability of failure detection) must be very close to 100 percent so that an undetected failure will be unlikely to occur and cause system failure.

SUMMARY AND CONCLUSION

This paper has presented the conceptual design of a distributed processor for general purpose computing and a method of analyzing parallelism in computations. Parallelism investigations have been shown to aid both the distribution of computations throughout a system and the actual structuring of a system. One of the motivations for developing the distributed processor was to take advantage of future SLI. In this regard, it was noted that future LSI development would make semiconductor memories competitive with magnetic structures, and, as a result, integration of a substantial amount of processing and memory on one wafer to form a cell would be practical. Arrays of cells using both local and global control were described and shown to offer high tolerance to failures, expandability, and the ability to efficiently execute general purpose computations.

When fully developed, the distributed processor should offer a powerful and versatile computing structure for special- and general-purpose computations. Its features should make it useful for a variety of applications from ground-based computation systems to avionics and space systems.

ACKNOWLEDGMENT

The research reported in this paper was sponsored by the Electronics Research Center under NASA Contract NAS 12-108. The authors wish to acknowledge help given by J. W. Hirsch on failure detection techniques for the distributed processor.

REFERENCES

1 D L SLOTNICK W C BORCH R C McREYNOLDS
The Solomon computer
Proc of the 1962 F J C C P 97

2 J HOLLAND
A universal computer capable of executing an arbitrary number of sub-programs simultaneously
Proceedings of the 1959 Eastern Joint Computer Conference.
3 L J KOCZELA et al
Study of spaceborne multiprocessing
Final report Phase I vol I and II—Technical Report Autonetics Division of NAA Anaheim C6-1476.10/33 April 1967
4 L J KOCZELA
Study of spaceborn multiprocessing
First Quarterly Report Phase II Autonetics Division of NAA C6-1476. 13/33 July 1967

BIBLIOGRAPHY

1 M RYBAK
Study to determine the applicability of the Solomon computer to command and control
Volumes I-IV report no ESD-TOR-64-184 submitted to U S Air Force L G Hanscom Field October 1964
2 B A CRANE J A GITHENS
Bulk processing in distributed logic memory
IEEE Transactions on Electronic Computers April 1965
3 C Y LEE
Intercommunicating cells, basis for a distributed logic computer
Proceeding of the Fall Joint Computer Conference 1962
4 H L GARNER et al
A study of interactive circuit computers
Report no AL-TOR-64-24 submitted to Air Force Avionics Laboratory Wright-Patterson Air Force Base Ohio April 1964

JOSS: 20,000 hours at a console—a statistical summary

by G. E. BRYAN
The RAND Corporation
Santa Monica, California

INTRODUCTION

JOSS* is a special-purpose computing system designed to provide users with a substantial and highly interactive computational capability.[1] The first JOSS system, developed for the JOHNNIAC computer machine by J. C. Shaw· was operational in early 1963. Work on an expanded system utilizing a modern PDP-6 computer began in 1964, and the system became operational in February 1966. Although there are many systems today that provide time-shared access to a computer, little is known of precisely how such machines are used. This was especially apparent at the beginning of JOSS development. Substantial effort was therefore made to provide a measuring or instrumenting capability within the system not only to record use of the system as a whole but also to record characteristics of use for individual users of the system. This paper presents the first results of these metering efforts.

To properly interpret these results, it is necessary to be familiar with the operation of the system, the features povided to users, and the hardware on which the system is implemented.

Figure 1 shows the essential hardware of the PDP-6 computer on which JOSS is implemented. The central processor is of modern design, typically executing instructions in 5μsec. A seven-level priority interrupt system is used for the control of input/output actions. Two independent 16,000-word core memories operate at 1.75μsec cycle time. A one-million-word drum is used for secondary storage of user programs. It has independent access to user core memory, can transfer 4000 words to memory in 17 milliseconds (4μsec each word), and rotates in 35 milliseconds. The organization is such that 60 milliseconds elapsed time is required to perform a complete user interchange between drum and core.

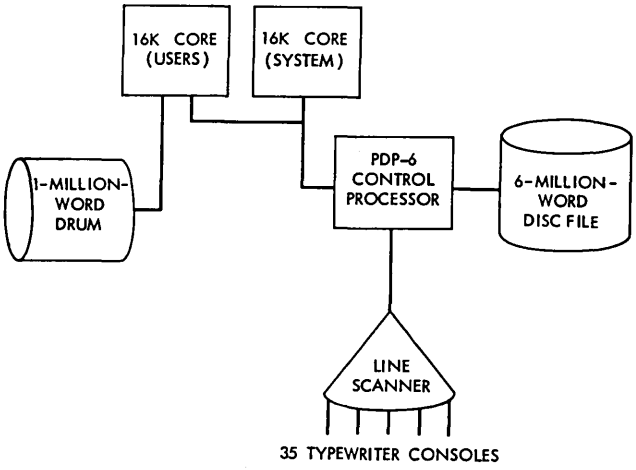


Figure 1—PDP-6 central processor

The transfer of information between drum and core is accomplished without interference in the use of core by drum and central processor references. No interference occurs because all software instructions are contained in a separate memory and thus have a maximum reference rate of once per instruction (about 4-5μsec) to the user memory. The maximum reference rates of drum and central processor to the user memory box, therefore, are easily accommodated by its 1.75μsec cycle time.

The disc file is a tertiary store for users' programs and data. Although it is possible for users to chain their programs by means of the disc, it is used almost exclusively for long-term storage, that is, at a low rate compared to its capability.

A line scanner that contains a one-character buffer in each direction, for each user's typewriter, connects each console to the computer. Interrupts from the scanner signal the complete transmission or reception of

*Joss is the trademark and service mark of The RAND Corporation for its computer program and services using that program.

each character. Currently, thirty-one specially designed typewriter consoles and six teletype consoles are connected to the system, although several times this number could be accommodated. Nine of the consoles are in remote locations operating over either private or dataphone lines, while the remainder are local to the computer with hard wire connections.

The system software, which consists of an interpreter for the JOSS language (the single offering of the system), is permanently resident in the system memory, together with software for I/O control and system supervision. Input/output for the users' consoles is buffered line by line through core areas in the system memory so that the user data need not be present in core during I/O delays caused by the consoles.

It is significant to note that no high-speed I/O devices such as tapes are available to users. Although the disc may be used for high-speed I/O, all data on the disc must have originated at typewriter consoles or have been generated by user programs.

System characteristics

Total usage of the JOSS system has increased rapidly since the PDP-6 version replaced the JOHNNIAC for regular JOSS service in February 1966, although the system has had its troubled times. Trends of total compute (central processor) time for users and the number of different users by month for the first year of PDP-6 operation are plotted in Figure 2 and point up several specific events. First, the activation of the disc and durm systems in July 1966 allowed a fivefold increase in the size of user programs and long-term storage of programs on the disc. Also, in the July-September period the number of consoles was almost doubled, from 18 to 30. Second, there were two sieges of machine trouble: The most serious lasted from late September through early November and a less serious one occurred in early February 1967.

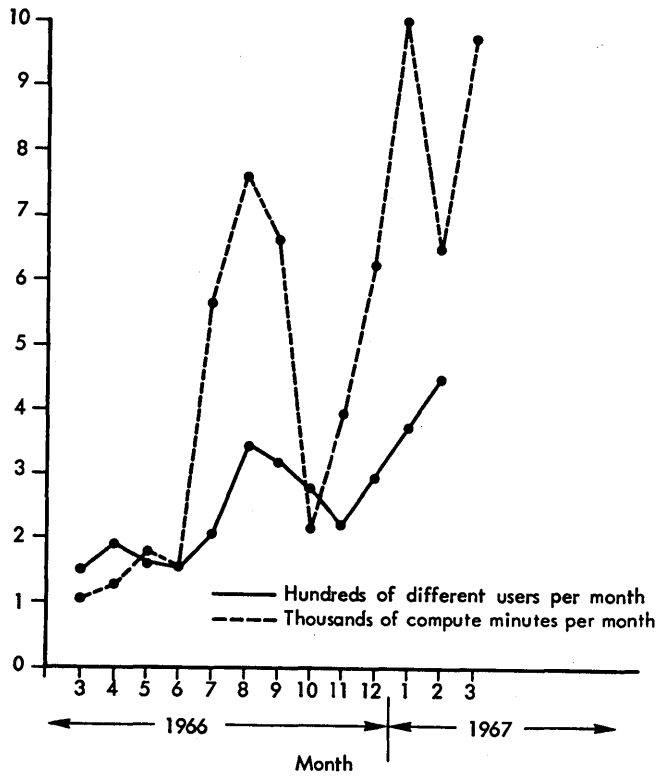


Figure 2—JOSS usage trends

It is clear that in a multiuser environment reliability is of critical importance. Every system failure directly inconveniences each user, and often, particularly in the case of remote consoles, users have very little idea of what has happened or when recovery is to be expected. This is in contrast with the batch mode of computer operation, where there is always the possibility of retrying the job and continuing without the users' knowledge. To provide information to users during times of breakdown, as well as to provide scheduling and other general information, the JOSS system includes as an adjunct a recorded telephone answering service.

Speed of operation is of particular interest in JOSS because it is an interpretive system. Figure 3 presents a distribution of the average rate of statement execution as measured each minute. The mean rate is about 10 milliseconds per statement, with the average statement including about five arithmetic or function operations. Both statement scanning and the arithmetic itself are interpretive, the operation time being divided approximately evenly between these two functions. The estimates from instruction counts within the interpreter, as well as measurements on specific problems, indicate that the interpretation process is slower than

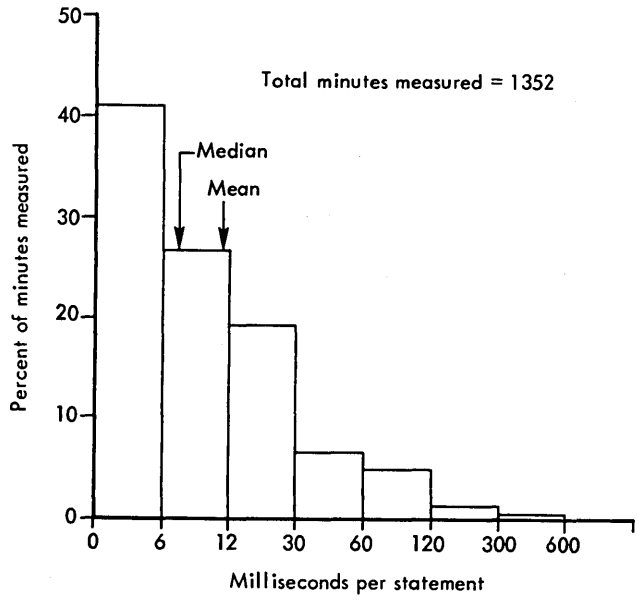


Figure 3—Statement interpretation rate

the equivalent compiled program by a factor of between 20 and 50.

Interpretation of a "typical" JOSS statement breaks down approximately as follows:

Interprogram step sequencing	1.0 ms
Execution of the action specified by the verb	1.75 ms
Five arithmetic or function operations (1.2 ms each)	6.0 ms
Interpretation time for typical statement	8.75 ms

In JOSS, statements may take far longer to execute than the times given above: for example, if they contain summations or nested functions of considerable complexity. However, compilation time which in many batch shops accounts for 25 percent of machine time, is eliminated. The elementary functions (e.g., *sin, log, exp.,* etc.) and I/O conversion run at speeds comparable to compiled code.

Program storage in JOSS is half that required for compiled-out code, while data storage requires twice the space. Individual programs can vary these ratios by large amounts. JOSS code is carried in character form. For example, the string $a = b + c$ occupies one word, while a compiled-out version would require three words in most computers.

Data, on the other hand, require two words—including the floating decimal representation and a link to associated numbers. Because JOSS stores only array elements that have a *defined value* rather than assigning block storage to arrays and because the system has the ability to operate meaningfully on sparse arrays, certain programs may require less total storage for data in JOSS than an equivalent FORTRAN program.

Average character I/O rates are shown in Figures 4, 5, and 6. The rates are typical for an aggregate of users during a prime shift. The overall ratio of output to input on a line basis is 2.85.

The maximum I/O rate is 15 characters per second per user, which is determined by the typewriter characteristics. At any given time, an individual typewriter can only be used in one direction (though the console does signal for certain synchronizing purposes in both directions simultaneously). Thus, in engineering the real-time console interrupt facilities, the maximum *total* rate per console is 15 characters per second per user. The real-time problem is actually less severe, because delays on character output can only result in slower typing and not in lost characters. Input characters can be lost, but for the average rates shown the probability is exceedingly small until several hundred consoles are attached.

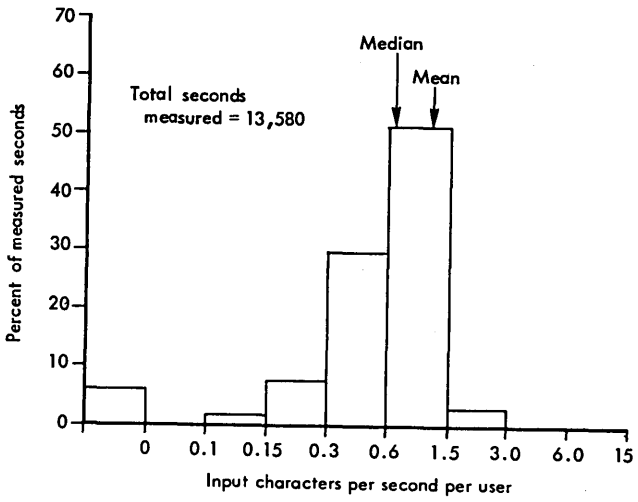


Figure 4—System input rate

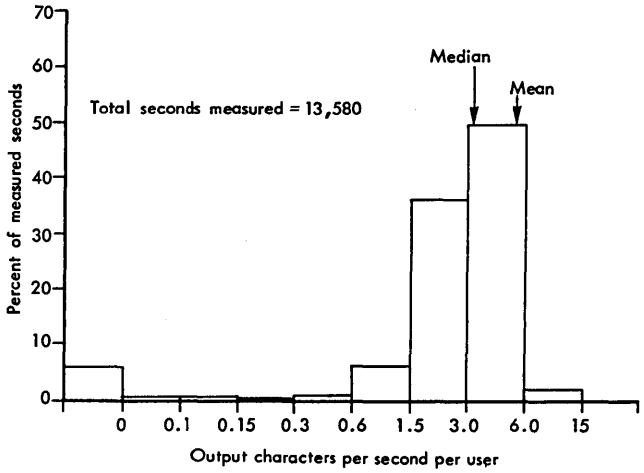


Figure 5—System output rate

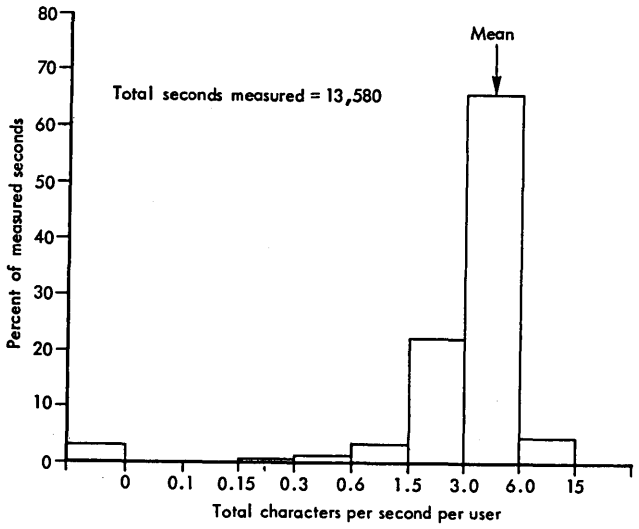


Figure 6—System input/output rate

The amount of core used, weighted by session time, is shown in Figure 7. The mean size is near 2000 words, and the shape of the distribution is similar to that measured in scientific FORTRAN batch-shop environments, although the median size in such environments is higher, namely, 4000 words.[2]

Program size is only a good measure of system load when weighted by execution time—large programs with only minor execution time do not provide substantial system load. In JOSS, when program size is weighted by execution time, the mean shifts upward from 2000 to 3000 words. The phenomenon of large programs tending to have longer running times has also been observed in batch FORTRAN environments. For example, an upward median shift from 4000 to 10,500 words was observed in one 1964 study[2] when the distribution of program size was weighted by program execution time. It is interesting to note that this same study observed *no* shift if only program instructions were considered. Thus, a more precise statement is that programs with a large amount of *data* tend to run a long time.

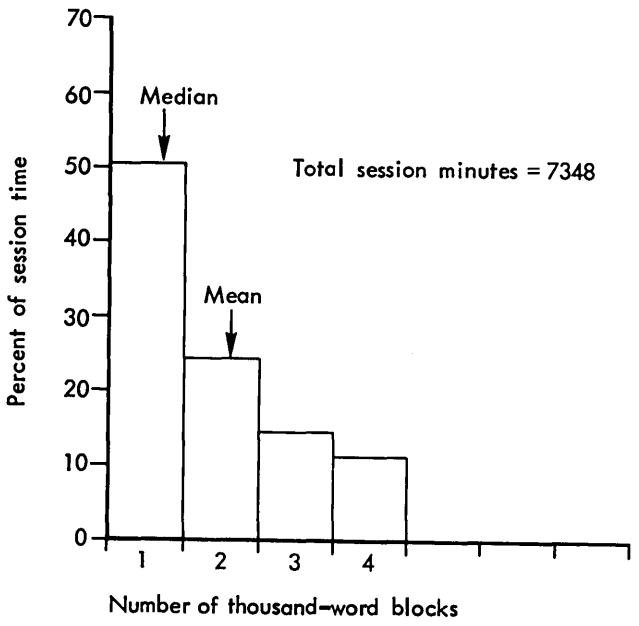


Figure 7—User program size

Overhead

Loss of computation power due to overhead functions is of concern on any computer and has been particularly critical in time-sharing systems. Modern equipment has eased the problem considerably by allowing simultaneous computing while performing swaps between high-speed memory and secondary storage. In JOSS, the limitation on user program size guarantees space for at least four user programs concurrently in core. Since variable-size blocks of from one- to four-thousand-word units are assigned to users, the maximum in core is sixteen. From eight to ten is a typical figure. JOSS suffers essentially zero loss of compute power due to memory interference from, or waiting for, swaps. JOSS supervisory overhead for user scheduling, resource allocation, performance metering, and accounting functions requires ¼ to ½ percent of real time. Moving users' data and programs within core memory to provide contiguous blocks of space for other users' programs being transferred from the drum, and checksum calculations for these transfers, are the major overhead items in the JOSS system. This is the price paid for variable-size user blocks in a machine without a paged memory. During periods of typical activity, overhead due to these functions uses about 5 percent of time, although brief periods (1 to 10 minutes) of very heavy activity may push the figure to between 30 and 40 percent.

Disc file usage

JOSS users maintain programs and data in the disc file on a long-term basis. Each user who so desires is assigned a file of up to 25 items, consisting, in aggregate, of at most 100 records of 128 words each. A user may have as many files assigned to him as he wishes. In March 1967, there were 330 files assigned to approximately 230 individuals and groups. Some large-user groups, particularly those outside RAND, operate with a pool of files assigned to them collectively: for example, the 100 students at the Air Academy.

In a typical session, users access the files about 5 times thrice to recall items, once to discard an item, and once to save an item. Thus, the total of 1000 actions each day, with perhaps 1000 words being transmitted in each action, gives an average rate of 40 words per second—a very modest load on the files.

The files have been in use since July 1966 and in March 1967 contained 9000 user records, which represents about 20 percent of the file capacity. In the first three months of 1967 the growth rate was steady at 1500 records per month, which if it continues, will fill the file in two years. Three factors may be responsible for this remarkably slow growth rate, which is at variance with experience in other time-shared installations. First, only one form of information is stored in the files —symbolic programs and data. In general-purpose systems, it is common for users to store absolute, relocatable, symbolic, and listing forms of the same program. Second, the limited size of the files and programs, together with the requirement to fill out a form (noncomputer) to get an additional file, tends to cause users to keep their files free of obsolete or no longer needed

items. Third, all input must come through users typewriters.

Characteristics of usage

Because JOSS is an interpretive system (in the machine a single program executes machine instructions that simulate the execution of JOSS language "programs"), there is a unique opportunity to measure the characteristics of both the JOSS users and the programs they generate. We can take advantage of this fact to record in detail many events—counting and timing statements· arithmetic operations, and user interactions. Even in the most expensive cases, counting actions require less than 1 percent of time.

A user session on JOSS spans the time from console turn-on to console turn-off. It corresponds to the log-on/log-off interval in most remote terminal systems. Figure 8 displays the distribution of session times for 227 sessions that occurred during one 24-hour period. The general form of the distribution remains quite constant over time with a mean in the range 45 to 50 minutes and median at 20 minutes. The length of the session may typify either the average computational needs of engineers and scientists or the amount of time it takes to solve problems that they are now willing to commit to JOSS.

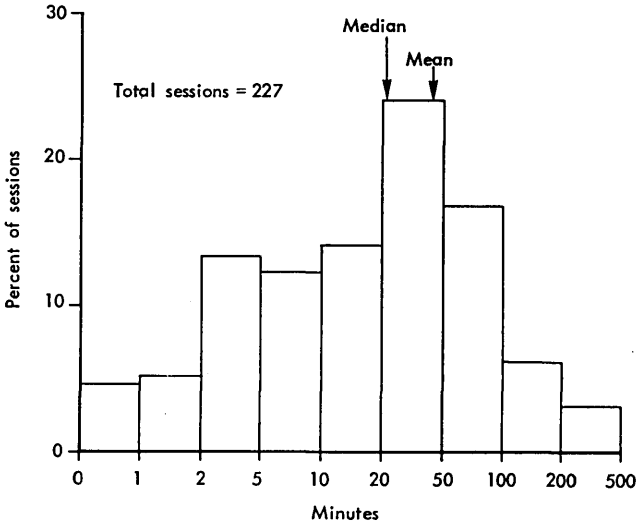


Figure 8—User session time

Program size, shown in Figure 9, is recorded at log-off time. Typical of the distribution is the peak at very small program sizes—10 cells will store only two or three program steps. The usage in this range represents the "desk calculator" usage of JOSS.

The mean amount of compute time used during each session is just under 4 minutes—in the same range as average compute times in scientific FORTRAN batch-processor shops. Figure 10 shows a typical distribution of compute time per user session. As in many computing environments, the mean compute time is misleading. For this sample, the mean is 6.6 minutes, while the median is 7 seconds, and 85 percent of all sessions use less than 50 seconds of computing. The general form of the distribution is typical: a high peak near the origin decaying rapidly at first and then very slowly leaving a long tail that contributes heavily to the average.

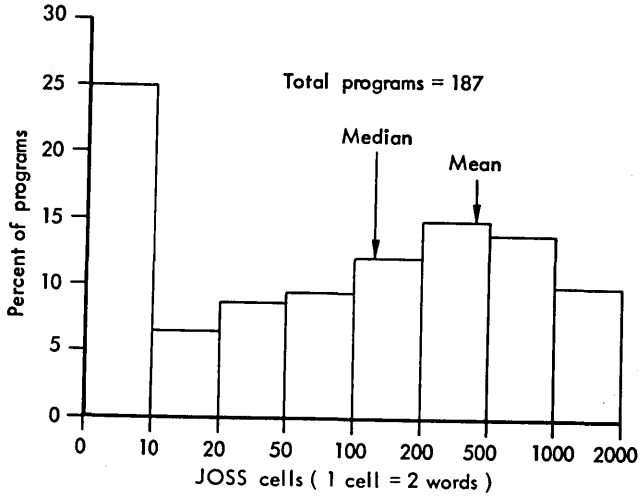


Figure 9—Used "size"

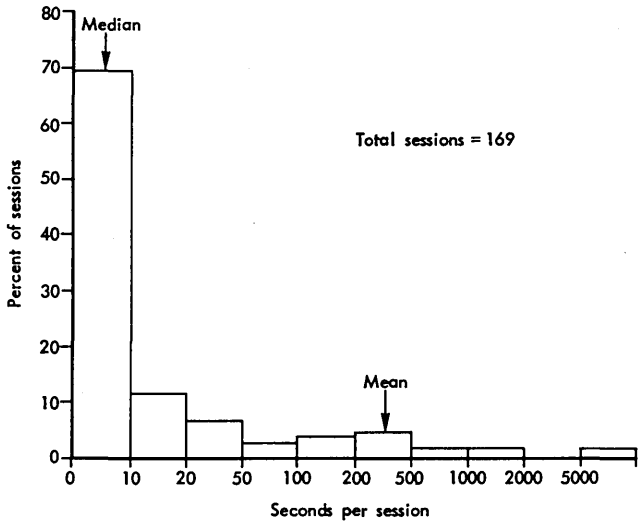


Figure 10—User compute time

The ability of a time-sharing system to respond rapidly to user requests depends heavily on the shape of this load curve. In the general-purpose, time-sharing systems,[3,4] typical load curves have far fewer jobs with small compute requirements and far more jobs with large compute requirements. This results in long queues

for compute resources and, in turn, in long delays in response. On the other hand, typical JOSS loads are such that the length of the compute queue seldom exceeds a length of three or four even during the heaviest usage hours.

For purposes of measurement, we divide a user session into subtasks, each initiated by the release of a typed command to the system. The requested tasks can be quite simple, such as *Type* 2 + 2., or more complex, such as those that initiate a computation of several hours' duration. During a typical session, the user hits carrier return 82 units, creating 82 tasks each involving, on the average,

1. Execution of 112 JOSS statements,
2. 500 arithmetic or function computations,
3. 2 seconds of compute time, and
4. 3 output lines.

As usual the averages are deceiving, as can be seen in Figure 11, which shows the distribution of compute time per task. The shape of the distribution is typical with a high initial peak and a very long tail. While the mean is 2 seconds, the median is 1/100th of that at 22 milliseconds.

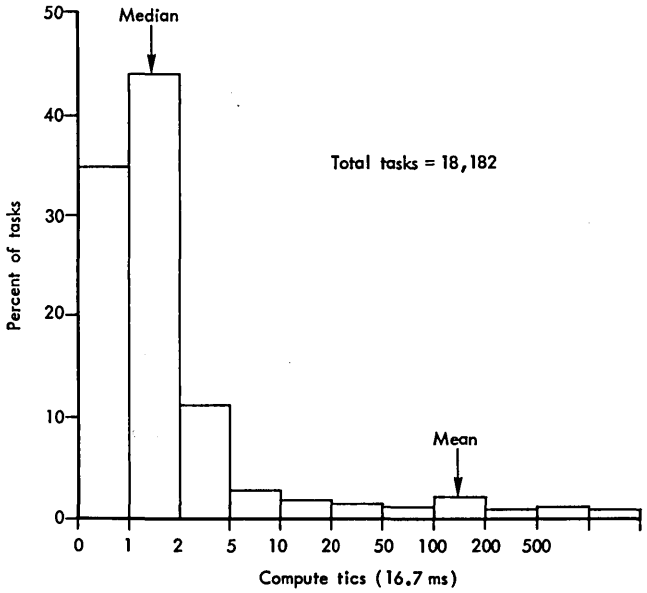


Figure 11—Compute time per task

The scheduling algorithm used in JOSS[5] gives highest priority to carrier-return interrupts, that is, to the processing of commands directed from the user to the system. Up to one time quanta (200 milliseconds) is allowed for the command processing before the task is considered compute-bound and relegated to round-robin processing in the compute queue. As can be seen from he compute request distribution in Figure 11, well over 90 percent of the requests are completed on a high-priority basis within the first quanta.

Task turnaround time is measured from the time a task is initiated until the last of any requested output is produced at the console—in JOSS the time from carrier return until console control is returned to the user. Figure 12 presents a typical distribution of task

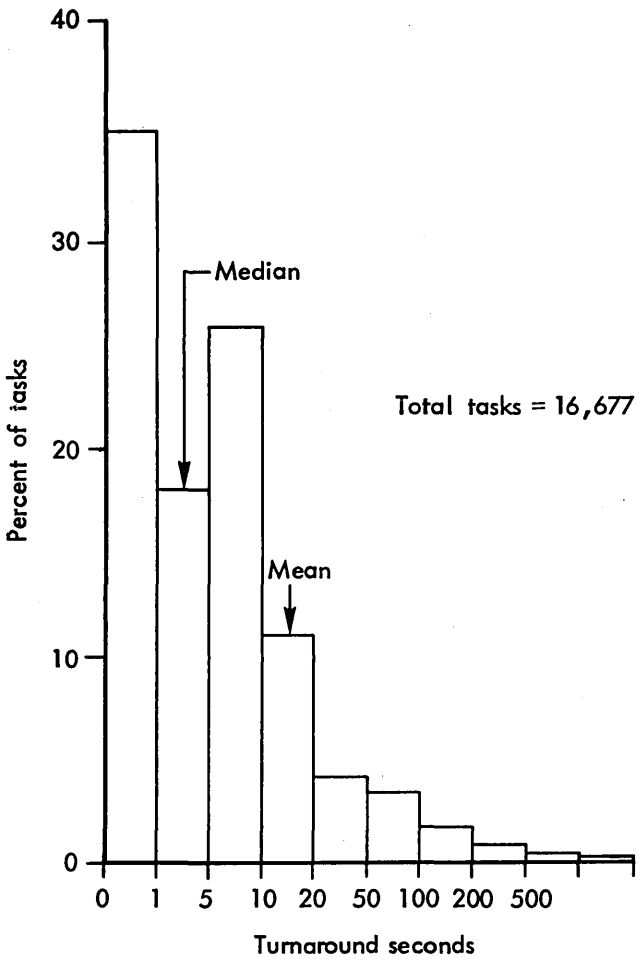


Figure 12—Task turnaround time

turnaround. Again, we have a mixture of a few rather long tasks with many short tasks. The mean turnaround time is 10 seconds including 7 seconds of typing and 2 seconds of computing. Some of the compute time is overlapped with the typing and some delayed because the computer is shared with other users. The *median* turnaround is 1.7 seconds; 90 percent of the tasks are completed in less than 10 seconds, the mean of the distribution. Usually, a user receives his answer with no noticeable delay between the time the request is made with carrier return and the beginning of the typed answer.

The distribution of time for the total user interaction cycle (the time between the initiation of successive tasks) is shown in Figure 13. The mean interaction time is 34 seconds and the median 11 seconds.

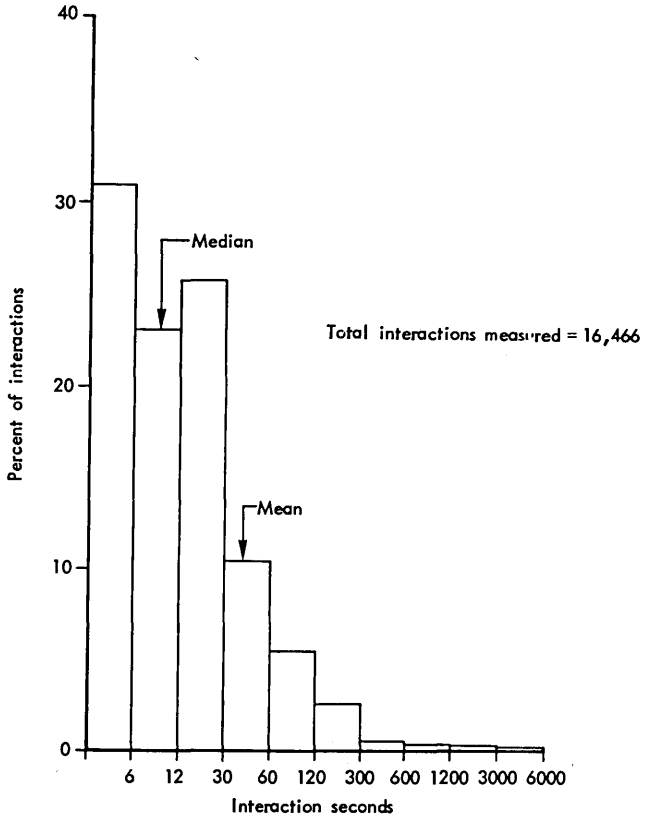


Figure 13—Interaction time

As lines of input and output information flow back and forth between JOSS and the user, the length of these lines, in characters, is recorded. Typical distributions are shown in Figures 14 and 15.

Peaks always appear in the output line-length distribution. The first one, between 0 and 5 characters, is caused by brief, commonly occurring error messages; the peak between 5 and 20 characters is the result of the output for *Demand* statements (request for input from the user); and other peaks, such as the one in the 30 to 35 range, are caused by specific programs producing a large amount of output during the measurement period, or by JOSS unformatted output.

We can now put together an interaction cycle that represents the "average" case. Since the median or "typical" case is so different in character, we also show that breakdown. In fact, in the real system we rarely see the average case because of the form of the distributions, while we often see the typical cases. Average and typical user interactions are shown in Figure 16. It is clear that a well-designed time-sharing system should be able to provide instant response for the typical user interaction even in the face of large

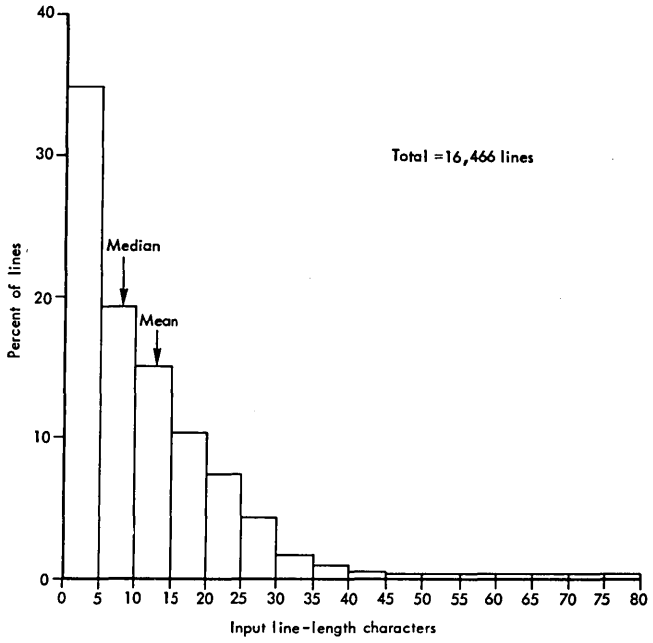


Figure 14—Characters per input line

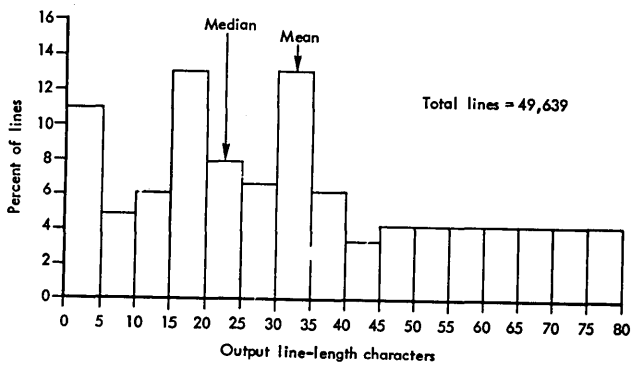


Figure 15—Characters per output line

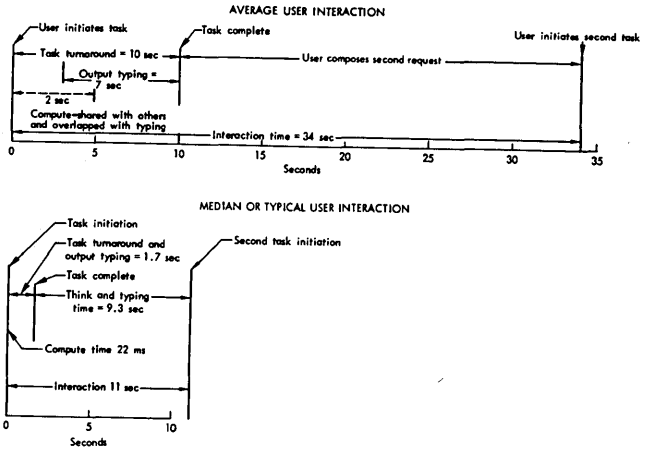


Figure 16—User interactions

amounts of compute time required for other users and any necessary swaps required to bring the user into high-speed memory. The average user, on the other hand, must take a delay of 1 second to receive his 2 seconds of computing.

CONCLUSION

The characteristics of a typical JOSS user session are as follows:

- Time at the console: 46 minutes.
- Compute time: 2.6 minutes.
- The user inputs 82 lines, creating a subtask for each line.
- For each task, 2.9 output lines are produced, the mean compute time is 1.85 seconds, but the median compute time is 22 milliseconds, and 90 percent complete in less than 154 milliseconds.
- Mean elapsed time between input lines is 32 seconds, distributed exponentially, and the output for the task is complete (task turned around) in 9 seconds. The median task turnaround time is 1.9 seconds.
- During the session, 15,000 JOSS statements are executed, and 68,000 arithmetic operations are performed.
- The user recalls 3 items from the files, discards 1 item, and saves 1 item.
- Mean program size is 650 words, but 50 percent are less than 200 words and 10 percent are larger than 2000 words.

The requests that users make of the JOSS system are substantially different from those made on general-purpose, on-line, time-shared systems. In JOSS, there are a relatively large number of requests for short amounts of computing and a relatively small number of requests for a large amount of computing—although the amount of computing is by no means trivial, as can be seen from the number of statements and arithmetic operations performed.

By providing a single easy-to-use language applicable to a wide class of problems *and* by strict limitations on program size and I/O speed (typewriter only, no tapes), JOSS has created a user environment in which very fast response can be given to typical user requests with simple I/O-based priority scheduling coupled to a simple round-robin scheduling of compute-bound users.

There are several distinctly different types of usage of this system:

1. Desk calculator.
2. Interactive problem solving.
3. Production computing.
4. Output-limited computing.
5. Multiconsole interactive games.

In the future, it may be desirable to report usage statistics by these types.

REFERENCES

1 G E BRYAN
JOSS: Introduction to the system implementation
The RAND Corporation Santa Monica Calif P-3486 November 1966 also published by The Digital Equipment Computer Users Society DECUS Proceedings Fall 1966

2 G E BRYAN
Dynamic characteristics of computer programs
The RAND Corporation Santa Monica Calif p 3661 August 1967

3 A L SCHERR
An analysis of time-shared computer systems
MACTR-18 thesis MIT June 1965 see also "Time sharing measurement" Datamation vol 12 No 4 April 1966 pp 22-26

4 R A TOTSCHEK
An empirical investigation into the behavior of the SDC time-sharing system
System Development Corporation Santa Monica Calif August 1965

5 G E BRYAN
JOSS: User scheduling and resource allocation
The RAND Corporation Santa Monica Calif RM-5216-PR January 1967

RAND Publications on JOSS

Baker, C. L., *JOSS: Console Design* (RM-5218-PR, February 1967)

————, *JOSS: Introduction to a Helpful Assistant* (RM-5058-PR, July 1966).

————, *JOSS: Rubrics* (P-3560, March 1967).

————, *JOSS: Scenario of a Filmed Report* (RM-4162-PR, June 1964); see also "The JOSS System: Time-Sharing at RAND," *Datamation*, Vol. 10, No. 11 (November 1965), pp. 32-36 (article based on RM-4162-PR).

Bryan, G.E., *JOSS: Accounting and Performance Measurement* (RM-5217-PR, June 1967).

————, *JOSS: Assembly Listing of the Supervisor* (RM-5437-PR, August 1967).

————, *JOSS: Introduction to the System Implementation* (P-3486, November 1966); also published by The Digital Equipment Computer Users Society, *DECUS Proceedings* (Fall 1966).

————, *JOSS: User Scheduling and Resource Allocation* (RM-5216-PR, January 1967).

Bryan, G. E., and E. W. Paxson, *The JOSS Notebook* (RM-5367-PR, August 1967).

Bryan, G. E., and J. W. Smith, *JOSS Language (Apercu and Précis, Pocket Précis, Poster Précis)* (RM-5377-PR, August 1967).

Gimble, E. P., *JOSS: Problem Solving for Engineers* (RM-5322-PR, May 1967).

Greenwald, I. D., *JOSS: Arithmetic and Function Evaluation Routines* (RM-5028-PR, September 1966).

————, *JOSS: Console Service Routines (The Distributor)* (RM-5044-PR, September 1966).

———, *JOSS: Disc File System* (RM-5257-PR, February 1967).

Marks, S. L., and G. W. Armerding, *The JOSS Primer* (RM-5220-PR, August 1967).

Shaw, J. C., *JOSS: Conversations with the Johnniac Open-Shop System* (P-3146, May 1965).

———, *JOSS: A Designer's View of an Experimental On-Line Computing System* (P-2922, August 1964); also in *AFIPS Conference Proceedings* (1964 FJCC), Vol. 26 (Spartan Books, Washington, D.C., 1964), pp. 455-464.

———, *JOSS: Examples of the Use of an Experimental On-Line Computing Service* (P-3131, April 1965).

———, *JOSS: Experience with an Experimental Computing Service for Users at Remote Typewriter Consoles* (P-3149, May 1965).

Smith, J. W., *JOSS: Central Processing Routines* (RM-5270-PR, August 1967).

How to write software specifications

by PHILIP H. HARTMAN and DAVID H. OWENS
Scientific Data Systems
Santa Monica, California

INTRODUCTION

In general, computer software is getting more complicated at an increasing rate. Unfortunately, the people who have to develop this software have been at it for only a relatively short time. The result is predictable: ever-larger software troubles. This, in turn, leads many people to feel insecure about software development, to think of it as a modern "black art" for which even the most able practitioners lose the recipe every few months.

We think many of these troubles are self-inflicted, the result of a natural desire to get on the computer (after all, isn't that what programming is all about?) instead of "wasting" time in planning and other "paper shuffling".

To illustrate this point, we particularly discuss the specifying of a FORTRAN compiler, selected because:

- FORTRAN is a well-defined language;
- it is well-understood because people have been using it for years;
- realistic comparisons can be made, based on actual compilers; and
- the sample of compilers is large enough to be meaningful.

Despite these factors, many people still have trouble turning out a good FORTRAN compiler within specification and time limits.

This paper is not a "cookbook" on how to write a compiler (that problem is too broad for a brief discussion), nor even what kind of compiler to write. Instead, it is an introduction to a way of thinking about and defining the product to be developed.

The user

Before you can write your specification, you must first define as clearly as possible the jobs your software must handle. To do this, consider every potential user or customer.

Eight separate types of users have been identified so far. The taxonomy is continuing, and other species may become apparent in time. Each user type has his own particular habitat and concerns. To write a specification that universally satisfies all their needs is almost impossible. However, it is important to identify the dominant types concerned with your system, and stress the particular features important to them.

The first user type is the *man of many decks*. He is easily identified by the state of his office, which abounds in printouts, paper tape festoons, and piles of punched cards that have run on some other computer system. Compatibility is his main concern. He wants a compiler to accommodate every program he has ever worked on and to solve every problem in every possible format.

The *speed demon,* on the other hand, is primarily concerned with execution speed. He wants a high-efficiency compiler that gives him fast-running object code. The new hardware with an instructions-per-second speedometer is particularly pleasing to him.

The *language purist* worries over syntax matters and language features such as floating-point DO loops or expressions in output lists. He wants to sit down and study all the nuances of the language, which can be a great advantage or a terrible hardship, depending on how elegant your language happens to be.

Most of us are familiar with the *klutz* species. Some computers must deal with many klutzes at once. (This is called "University Time-Sharing", or UTS for short). Engineers, as well as students, often fall into this category. When you know that your system must accommodate these unseasoned or "non-user" users, you have to pay more attention to diagnostics, making sure they give clear warning and error messages. The primary

concern here is the ability to get new programs written and debugged quickly. Syntax relaxations are important, so the software can minimize the "arbitrary" bugs that often occur with sensible but not-quite-proper usages. An automatic debug feature that allows debugging in the source language is valuable for a klutz-oriented system.

The *bit-chaser* is at the other end of the spectrum. He wants to study the object code listing and attempt new feats with the language. If you specify a system that lets him use in-line machine code, making the compiler behave like an assembler, he can kludge around in machine language to his heart's content.

The *real-time cat* wants to run programs in real time, and tie them to interrupts. Therefore, object code reentrancy is important to him. He also values language features for dealing with interrupts.

The *pioneer* is one of the nicest users to work with. He is primarily interested in solving his problem, and responds with knowledgeable enthusiasm to new features. He reads specifications carefully and makes positive suggestions. He is patient with bugs and delays, kind to programmers and his mother, and, unfortunately, a rather rare species.

The true *time-sharer* is a new breed, but proliferating rapidly. For him, you should consider a desk-calculator mode that lets him evaluate expressions. The automatic debug feature (which allows debugging in source language) is important to him; he also appreciates text-editing features so he can correct his source program easily. But (unlike the klutz) he knows what to do with modern machines, and makes strenuous demands for software that is as flexible and up-to-date as his hardware.

Every FORTRAN system serves one or more of these user types. To specify software that can satisfy their often-conflicting needs, you must identify your users, then pin down their needs (both stated and unstated).

The software

A surprising number of things need to be specified in defining a piece of software. The following section discusses some of the more important ones.

Syntax

Syntax is usually defined by the manufacturer to a greater extent than any other software feature. A syntax specification defines legal and illegal usages and explains the meaning of legal usages. Syntax is usually specified in a reference manual, often published in advance of the delivery of the software. We contend that a syntax definition (such as the USASI* standard) is only the *beginning* of a software spec.

*United States of America Standards Institute, formerly ASA.

Diagnostics

Three facets of the diagnostic problem should be considered: compilation diagnostics (produced by the compiler), execution diagnostics (produced by the library or run-time package), and somewhat separate from the diagnostic messages themselves, the recovery taken by the system; that is, once the system has informed the user of his indiscretion, what happens?

Compilation diagnostics. Should the messages flag the exact character at which the error occurred? (Many compilers do, but this is seldom specified.) Should the error messages be interspersed with the listing of the source program, or simply summarized at the end (sometimes difficult to correlate with the source program)? Certain errors (such as undefined labels) probably have to be given in a summary rather than interleaved with the source listing. Which errors will be summarized and which interleaved?

How detailed will the summary be? For example, will doubly defined labels be distinguished from undefined ones, or will they all be lumped together under "LABELING ERRORS"? Will the summary also list the line numbers on which offending labels were used?

Can you suppress the source listing but still get the diagnostics? This is particularly important to users without line printers. They can't afford the time for a complete listing, yet they can't ignore the diagnostics.

How detailed will error messages be? This has two aspects: what errors will be detected, and which will be distinguished from one another. Consider, for example, a compiler that detects every conceivable thing that can go wrong with a source program, but has only one error message—something illuminating like "ERROR". At the other extreme would be a compiler that detects only two errors, diagnosing them superbly with messages such as "MISSING COMMA IN COMPUTED GO TO STATEMENT" and "SUPERFLUOUS COMMA IN DO STATEMENT", meanwhile ignoring every other possible error.

Every FORTRAN source program has a vast potential for possible errors. Detecting and reporting these can range from trivial to horrendous. Keep in mind the level of diagnostic detail you'll need for the compiler, and help your implementation group by specifying at least a few guidelines in this area.

Execution diagnostics. Quite apart from error detection in the compiler are execution-time diagnostics. For example, what happens when you try to take the square root of a negative number? Is the message you get (if any) related in a reasonable way to the source listing of the program, so you can figure out where the problem is? What about bad characters in input records or errors in FORMAT statements? Are execution diagnostics coordinated with a debugging package?

Execution diagnostics are more adamantly ignored by designers than compilation diagnostics, and there is much room for improvement here.

Error recovery. When the compiler detects and reports an error, what happens? Is the entire compilation aborted? Perhaps the compilation continues and the offending statement is deleted. Small compilers often throw out a statement once they have found a single error. Larger compilers often try to find more errors in the same statement; in such a compiler the error messages should indicate whether or not a statement has been deleted, since some errors cause deletion and others do not.

What happens to a statement with an error? Some compilers generate partial code that won't work. Some delete the offending statement and proceed as if it had not appeared. Others replace a deleted statement with a call to a library routine that aborts the job.

To what extent can errors in one statement introduce errors in other statements? When a conflict in the attributes of a name is detected (such as an attempt to declare a variable both fixed-point and floating-point), some compilers flag and delete *all* statements that use that name in any way. This might confuse and frustrate the user, who may prefer simply throwing out the later declaration that caused the conflict.

At execution time, does taking the square root of a negative number cause the job to abort? Or is some fairly reasonable answer given so that the job may continue? Can the user control how many times this particular error is to be detected before it *does* abort the job? (Ten thousand SQRT messages can produce a lot of paper.)

Debugging features

What features are you going to provide so the user can debug his own programs? A popular item these days is a set of tools for debugging a FORTRAN program in source language.

For example, it is nice to be able to trace the execution of a program at the statement level, finding out which statements were executed and in what order, and (in some cases) what values were computed by the various statements. There are often trace programs that trace the machine instructions generated for a FORTRAN program. However, since many instructions are typically generated for each source statement, such programs are not convenient for the FORTRAN user—especially one who does not understand machine language.

Trapping is also useful; it is nice to be able to say "Execute my program without printing a trace, but when you reach line 23 of subroutine GRUNCH, stop, and I'll figure out what to do then". This facility is analogous to the Address Stop switches on some computers. Or, you may want to trap every time the variable XYZ is changed, an ability comparable to Effective Address Stop switches. Or, you may wish to detect subscript values that are out of range.

These capabilities are generally handled by options whereby the compiler inserts extra code into the object program; this code consists primarily of calls upon special library subroutines that perform the tracing and trapping. This appeals particularly to users in the "klutz" and "pioneer" categories. A debug package specification should define the compiler options and illustrate the code generated by the options. It should define the output produced by the library routines and describe the operating procedure for using the package at execution time. The specification should also explain how the debug package is to be coordinated with execution diagnostics.

Sticky wickets

A sticky wicket is an obscure usage that can cause the system to do strange things. Under what circumstances can input to the compiler, however unintelligible, cause the compiler to halt, hang up for indefinite periods, or otherwise fail to give a reasonable diagnostic and proceed? For example, a statement like "X = 1.E8000000" (which could result from a keypunching error) has been seen to hang up a compiler for hours.

What happens at run time if a FORMAT statement that contains no format codes capable of transmitting data (i.e., no E, F, I, A, etc.) is used with an I/O statement that transfers data? Since the processor normally re-scans a FORMAT statement when data remains to be transmitted, what is to prevent it from looping indefinitely?

Sticky wickets are hard to anticipate. Yours will probably be different from ours.

Object code listing

The format of object listings is often overlooked when compiler specs are written. As a result, many listings are hardly readable. To begin with, should there be one? In a small compiler, you may not want it, particularly if the generated object code isn't very impressive. If there is one, do you expect it to be readable? If so you should consider such points as:

- Are the source lines interspersed in the object code?
- Are the user's statement labels shown? If neither this nor the above is done, then it is hard to relate the object code to the source statements.

- Are the instructions listed mnemonically, with the same mnemonics used by the assembler?
- Are variables, subprograms, dummy arguments, etc. referenced by name, or is it acceptable to merely use their locations?
- Are constants (integer, complex, logical, etc.) referred to meaningfully by value? Many compilers list them in octal, or list only their locations.

Compilation summary and load map

The compilation summary, usually printed out at the end of compilation, gives the programmer information about his program, e.g.:

- What names he has used in his source program, what the compiler thought they were (variables, arrays, subprograms, etc.), and their sizes, types, and storage allocations.
- Size and composition of the various sections of the program, such as generated code, constants, temporary storage, local variables, COMMON, etc.
- Number and severity of errors.

A compilation summary is particularly useful as a debugging and documentation aid. No two compilers do the summary in quite the same way. Some are brief and vague; some are long-winded and vague; and some are well-designed and useful. The latter type takes some work on the part of the implementer, so to achieve the desired result you must specify it.

The load map is a printout showing where program segments are loaded. In debugging, for example, this information is often necessary. The map generally includes all the user's subroutines as well as the library routines he needs. Should it include the user's *main* program (which may be his *only* program, and is certainly the most important part to him)? Should the table include labeled COMMON blocks? Should it define the limits of blank COMMON? Should it include the size of each subroutine and data block? Should the total used memory be printed?

Should the load map entries be in some sensible order? Usually they are shown in the order they were loaded. Thus, for example, the size of a program can be determined by subtracting its origin from that of the next entry. Some loaders, however, order the entries in other, seemingly random, ways. Unless the sizes are printed, they are then quite hard to determine. It is also hard to answer the question "In what subroutine does this location lie?"—a query that arises often in debugging.

If the program overflows memory, should the loader continue to simulate loading and thus be able to print out *how much* too big the program is? This feature, which can also be considered under recovery from errors, is useful.

And finally, should there be a load map at all? Perhaps, in small configurations, it is preferable to save space in the loader.

Monitor interfaces

There is a natural tendency to put off consideration of monitor interfaces until shortly before the system is delivered. By then it is rather late to correct the things that have gone wrong.

The specification should discuss features such as the following:

- *Binary object language*—how does it handle such FORTRAN features as NAMELIST, blank COMMON, labeled COMMON, the DATA statement, the distinction between subprograms and main programs, and the size of programs? (Is size given on the first or last record of the program?) This last item can drastically affect loader design and efficiency. It's also important to know what the assembler does in this regard.
- *Overlay or chaining*
- *I/O device assignments*—what are the conventions at compilation and execution time? How do ASSIGN cards work? How does the user assign logical unit 108? What happens if he does not assign it?
- *Vertical format conventions on printed output*—are these implemented in the monitor or in the FORTRAN library? (Many factors favor the monitor.)
- *Typewriter input/output*—can the typewriter be assigned like any other I/O device, or are there strange restrictions connected with it? Must the FORTRAN library worry about the typewriter as a special case?
- *The PAUSE statement*
- *Error interface*—how does the compiler report to the monitor that it has detected errors? Can it report the severity of the errors? Can an error detected in compilation prevent execution in a compile-and-go job? Can the user over-ride this if he wishes?

Maintainability

The compiler specification should consider features to assure the maintainability of the software, and also improvement after it is in use.

Assembly language. What language is the compiler written in? Can it be assembled on the machine on which it runs? Some compiler-writers use their own assembler during implementation (especially if the assembler for the target machine is being developed concur-

rently). Must the compiler be maintained that way or can it be converted to the new assembly language later? Whose responsibility is the conversion?

Can the compiler be assembled on the same *size* machine it can run in? Small, well-parameterized compilers tend to use a lot of symbols, which may make them impractical to assemble on small configurations. You may want to specify that the compiler be modular, with each assembly segment relatively small. Then again, this may simply postpone the problem until load time.

Compiler debugging package. What aids are to be provided for the debugging of the compiler itself? You may say "What difference does that make, as long as it's debugged when I receive it?" But *you* have to maintain it. There will be bugs that show up later, and you may want to add improvements, which will have to be checked out. A large compiler is a complicated program, and debugging it without a high-level debug package can be unpleasant.

Naming of system routines. In addition to the library routines with standard names (such as ABS and SIN), a FORTRAN system contains many subroutines used to process the language, such as:

```
9REWIND   (rewind statement)
9RTOI     (real-to-integer conversion)
```

What are these routines to be named? Should they be descriptive, or is it acceptable to name them anything at all, such as:

```
BF:ST     (rewind statement)
L:23R     (real-to-integer conversion)
```

For compilers that produce an object code listing, it may be desirable to print names the user can figure out, for much the same reason you list the double-precision divide instruction as something like DPD, rather than ADD (Arithmetic Double Divide).

Documentation

Will a narrative description of the compiler's workings be provided? An adequate one might run to 500 pages. Will flow charts be provided, and if so, to what level of detail? What notational conventions will insure the readability of the flow charts? Will the comments on the compiler listing be adequate? Who will decide whether they are, and how?

Aesthetics

Many important facets of a compiler have no particular organizational place to be specified, yet they still affect its over-all usefulness. The category of aesthetics is particularly hard to defend because it consists of "non-essentials." The lack of most features we've already discussed will make something impossible. (For example, the lack of a debug package makes it impossible to debug in source language.) The lack of an aesthetic, on the other hand, doesn't usually make anything impossible—merely unpleasant.

Attractiveness of listing and summary. When the compiler produces a source program listing, is it pleasant and easy to read? Is it always possible to distinguish source lines from remarks printed by the compiler? Are the source lines numbered? Is the listing single or double-spaced? (Many users object to the bulkiness of a double-spaced listing.) Does it have a clean, uncluttered appearance?

Is it possible to misconstrue the compilation summary as part of the source listing? Do unnecessary page ejects between categories in the summary produce many pages with very little on them? Are categories within the summary arranged in a logical order?

Ease of use. Is the combination of FORTRAN and the monitor easy to use? Are the required control cards few in number and easy to understand? (One well-known Operating System has earned the reputation of being very difficult to use, requiring an expert even to load the monitor. Such a reputation does not help a product's marketability.) Is the error recovery well-designed, or does the system have a small tantrum every time something goes wrong?

Consistency with previous systems. Is the operating procedure similar to that of other systems with which the user may be familiar, or must he learn a whole new procedure? If it is different, is there a good reason (such as extended capability), or is it simply the result of bad coordination? Are the control cards similar to those the user knows, or do they have deceptive differences (such as one system in which the LO option means **L**ist **O**bject code, and another in which it means **L**ist the s**O**urce program)?

Aesthetic features are, for the most part, non-essential. But they are not unimportant. Users can become very emotional about them.

Compilation speed

This is sometimes mentioned in specifications. For example, a compiler specified by one manufacturer was to compile 4,000 statements per minute. Ignoring the fact that such figures are usually plucked out of the air, what does this "4,000 statements" mean? Does it mean 4,000 comment cards? Assuming it means full-fledged statements, are they statements like

```
X = Y
```

or statements more like

```
WRITE (108,667)  ((TDEL(I, J), AGRZ(1-1,
2 J-1), AGRZ(2*I+7, J-1), RHO(I, J), I=
5 OLPPTR(J), AVGFLO(J), J=1, NVALS),
```

```
4 YMAX(J), ZMAX(J), J, NEWPTR(J),
5 OLPPTR(J), AVGFLO(J), J=1, NVALS),
6 NVALS, PTOTAL, (TEMP(K), K=1,NTEMPS),
7 NTEMPS, LAST, ESTVAL
```

The latter is seven cards long, but is a single *statement* according to the USASI FORTRAN definition, which allows statements of up to 20 lines. Thus, 4,000 statements a minute could mean 80,000 lines a minute, each as involved as those above.

To give this specification meaning, you should develop some standard test programs of "normal complexity (whatever that may mean to you.) These programs should be part of the specification, with remarks like "This program should compile in 15 seconds."

Effect of peripherals on speed

Before defining speed criteria such as above, consider carefully what compile options and I/O devices will be used during the tests. If you time a benchmark compilation while getting a complete object code listing on the line printer, you are simply measuring the speed of the printer, not the compiler. Instead, you need to measure compilation speed under conditions that minimize I/O time, e.g., with no listings and input from tape. Measurements with other devices can also be included.

Note also that if your system offers no input device faster than 1,000 lines per minute, you are wasting your money to develop a compiler that goes 4,000 (assuming no time-sharing).

Job overhead

To some users, job overhead may be more significant than compilation speed. A specification might require that once the compiler is in the computer and chugging away, it could compile 1,000 lines in 15 seconds. But what if those 1,000 lines are broken up into 20 subroutines, each requiring a separate compilation?

In many current systems, such a change increases the compilation time by an order of magnitude because the fast compilation is swamped by monitor overhead. A good many FORTRAN jobs are quite short: under 50 source cards and under a minute of execution. For such jobs, system overhead can be more important than compilation speed.

Object code efficiency

There are two aspets to object code efficiency: speed and space. Up to a point, the two go hand in hand—the shortest object program is the fastest. As you tend toward more efficient object programs, the trend reverses—small space works against high speed, and vice versa. The most widespread concern recently has been for speed, not space.

A compiler can perform two kinds of optimization: local and global. Local optimization considers only one statement at a time, without regard to neighboring statements. Some examples are:

- optimizing negation within an arithmetic expression;
- evaluating X**2 (X squared) with multiplication instead of calling a general routine that can raise X to any power;
- taking advantage of special instructions, like using a Memory Increment instruction for the statement "J=J+1"; and
- generating clever code for logical expressions.

Global optimization considers several statements, or perhaps a whole program, at a time. A globally optimizing compiler looks at a DO loop and all the statements within it to find those subscript expressions that depend on the value of the induction variable. It then tries to use index registers to handle subscripting efficiently throughout the loop. It may observe that the expression X+Y/Z is used in two different statements and that X, Y, and Z do not change in between. It then evaluates X+Y/Z only once, and simply keeps the result around for the second use.

Object code speed is difficult to quantify. It is seldom possible to say "This compiler produces code that runs twice as fast as that compiler." It is much more likely that Compiler A produces faster object code for one test program, Compiler B for another.

Thus, a compiler specification needs a narrative description of the local and global optimization it is intended to perform. This narrative should be liberally sprinkled with examples of source statements and the object code produced for them. The number of examples needed depends on how elaborate the optimization is. For example, if the compiler generates code for all DO loops the same way, only one example of DO loop code need be shown. If the compiler distinguishes five flavors of DO loops, then the differences should be explained and code examples shown for all five.

Such a description can be judged adequate if, after reading it, a reasonably competent person (such as yourself) can figure out with pretty good accuracy what the object code for any given statement will be. Though the number of possible source statements is practically infinite, the object code description needn't be terribly long—perhaps 10 to 30 pages for most compilers.

For comparing object code speed, you should develop several FORTRAN benchmark programs that test the speed of various kinds of object code. *One* pro-

gram selected at random is seldom very comprehensive. We received one recently that tests DO loops and subscripting quite well, but completely ignores the speed of the math library (it doesn't use any functions), common subexpression elimination (there aren't any common subexpressions), logical expressions, etc. Performance of future compilers can be related to a *set* of such benchmarks.

It will not be hard to determine execution time of these test programs, even before the compiler is written, given the object code description and the instruction timing of the machine. It is often possible to ignore whole sections of the source program because they don't contribute much to execution time. For example, the benchmark above is some 30 statements long, but 95% of the execution time is spent in just two statements. This program may be an extreme example, but it is quite usual that the bulk of the execution time is spent in a small part—the "inner loop".

Some things that should be discussed in the object code description are listed below.

Subscripting

How does the compiler handle:

- constant subscripts
- single-variable subscripts
- multi-variable subscripts
- subscripts on dummy arrays
- subscripts in calling sequences
- subscripts in DO loops
- subscripts not in DO loops
- partial-word addressing (e.g., halfword, byte)
- multiple-word addressing (doubleword, tripleword)

Subroutine calling

Describe:

- calling sequences
- receiving sequences
- argument passing
- timing considerations of system subroutines involved with calling, receiving, and argument passing
- subscripted arguments
- arguments which are themselves dummies
- register-saving conventions—which registers may a subroutine destroy with impunity, and which must it preserve? (This information is needed by the machine-language programmer and it also sheds some light on object code timing.)
- system subroutines — assuming that the points above relate to subroutines that can be coded in FORTRAN, what about the supporting library, which must be coded in machine language (e.g., I/O, PAUSE, X**Y, complex arithmetic)? Describe calling sequences, timing, register-saving as it relates to these.

Arithmetic expression evaluation

- how are +, —, *, /, and ** handled?
- what about INTEGER, REAL, DOUBLE PRECISION, COMPLEX?
- mixed mode (if allowed)
- function arguments—are they evaluated before or during expression evaluation?
- subscripts—are they evaluated before or during expression evaluation?
- sign optimization—under what circumstances is negation performed in arithmetic expressions, and what steps are taken to avoid it?
- constant subexpressions (e.g., 13.7+4.8/17) — are they evaluated at compilation or execution time?

Arithmetic IF statements

- special-casing when certain labels are equal
- jumps leading to the following statement
- INTEGER, REAL, DOUBLE PRECISION types
- is subtraction always used to evaluate X—Y for comparison with zero, or is X sometimes compared with Y?

Logical expressions

What does the compiler do about:

- logical IF statements
- logical assignment statements
- logical IF statements controlling GO TO statements
- relational operators
- .AND. and .OR. (and .EOR. if allowed)
- .NOT.

DO loops

- by themselves
- considering the subscript expressions within them
- jumps into and out of loops
- materialization of the induction variable
- constants and variables as initial, terminal, and increment values for the loop
- special cases

Common subexpression elimination

- within statements
- across several statements
- across program flow
- moving expressions out of loops (both DO loops and loops implied by the flow of the program)

- ABNORMAL functions
- ABNORMAL variables
- multiple register optimization
- effects of COMMON, EQUIVALENCE, dummy arguments, and the like
- thoroughness in discovering common subexpressions (e.g., is A+B the same as B+A?)
- sign optimization (e.g., is —A—B shared with A+B?)

Intrinsic functions

- which are recognized by the compiler?
- which generate special in-line code?
- what in-line code do they generate?
- which call special subroutines (like 9SIN on the SDS 9300)?
- sign optimization (e.g., does the compiler know that SIN(—X) = —SIN(X)?)
- common subexpression elimination (e.g., does the compiler know that MAX(J,K)=MAX(K,J)?)
- special cases

System capacity

This characteristic can be summed up in the question "How big a program will it compile, load, and execute?" Naturally, the answer depends on how much memory the machine has, but it is necessary to have standards of performance for the smallest configuration, plus some indication of how performance will improve as the configuration expands.

Compilation. Compilation capacity is often measured in terms of how many source statements the compiler can handle before it runs out of memory. This is misleading, because the limiting factor is not usually the number of *statements* (most compilers can compile unlabeled CONTINUE statements forever—or at least until PL/I is implemented) but the number of *names* and *statement labels* used. The compiler must keep these tables from the first encounter until the end of compilation; hence they consume memory. The space required for individual statements is generally released at the end of the statement.

Long programs typically do have more names and labels than short programs, but there is no simple rule for saying how many more. We have heard it said that a given compiler could handle a "typical 400-statement program", but no definition is ever given for such a program. To define this "typical" program may require three or four 400-statement benchmark programs, each covering a different area. One could be heavy on statement labels, another on array declarations, another on COMMON and EQUIVALENCE, etc.

Execution. Some FORTRAN systems have been developed in which the compiler performs admirably in a small configuration, but no object program will fit at run time. The space at execution time is sliced up more ways than at compilation time: some is taken by the resident monitor, some by the FORTRAN run-time and/or library package, some by the object code produced by the compiler, some by the data storage, and some, possibly, by the loader.

The space for data storage is generally dictated by the hardware (e.g., one word for integers, two for certain floating-point variables, etc.). The space required by the object code depends on the efficiency of the compiler and on the hardware. In few cases does a small change to the compiler have a large effect on the size of the object code, assuming that the compiler is reasonably well written. The run-time and library package, the resident monitor, and the loader are most amenable to control by the system designer.

Aside from writing short code, the main consideration in designing a library is: must the user load a lot of routines he doesn't need? To avoid this, a library is generally segmented into many different load modules (separate assemblies), and only the ones needed are loaded. Placing several subroutines in the same assembly may simplify things for the library programmer, but it hurts the FORTRAN user. The complier specification needs a remark discouraging this kind of clumping together of library routines: It isn't practical to have a benchmark deck simply to check whether this has been done or not, but you *can* check for it by inspecting library listings.

Benchmarks are useful to help determine how much space is required to support certain "typical" programs. For this purpose, it is useful to develop a set of small (20 lines or less) programs, some using I/O statements, thereby calling the library I/O package (which is usually large and is required by virtually all FORTRAN programs), some using floating-point (especially on systems with programmed floating-point), some with mathematical functions, etc. For each test deck, the specification says "The object code for this program should not exceed X words, and the library required to support the program should not exceed Y words." Be sure to take into account the kind of system you're designing for, since the object code figures can vary widely according to whether basic operations are done with subroutine calls or machine instructions.

Finally, since execution is often more of a bottleneck than compilation, execution capacity ought to be fixed first, then a compiler specified that can compile programs of that size in the same configuration. This

approach might halt the tendency to cram compilers into needlessly small spaces.

Loading. It is a natural assumption that if a program will execute in 8K, it can also be loaded in 8K. The amusing situation sometimes develops, however, that the program would run if it could be loaded, but it won't load. This is partly because a loader must keep tables of external symbols to be linked together during loading; these tables are not needed during execution. Some loaders avoid this problem by putting tables in the area that will become blank COMMON at execution time (admittedly, this solution is no help in programs that don't use COMMON).

Sometimes the problem is that the loader itself takes up space that is not recoverable at execution time. Some systems get around this by having a run-time package. This package is the portion of the library that is needed by almost every FORTRAN program. It typically includes the I/O package and the floating-point arithmetic package (for machines without hardware floating-point). It is assembled in absolute form, and takes only a small bootstrap program to load. The run-time package is designed to be the same size as the FORTRAN loader, and the loader's last act is to commit suicide by invoking the bootstrap and loading the run-time package on top of itself.

Other problems may result from not knowing size at the beginning of a load module. This can force the loader to slide the program around in memory or make multiple passes. If such manipulations are necessary, this should be pointed out in the specification.

Mechanics of using the system

Suppose the user has learned the language and written a program, and now wants to step up to a machine and try to run and check out his program. How does he load the compiler, specify options, load his compiled program, and load the library? In a monitored system there may be monitor specifications for this, but not in a free-standing system. This becomes particularly important in paper tape systems, which are basically cumbersome to begin with.

A properly designed and documented system can minimize the number of runs required to get a user's program to work. These topics are normally covered in a FORTRAN operations manual rather than a language reference manual. This kind of documentation has typically not been written until the system was finished, if at all.

Consecutive compilations. It has been standard, though unspecified, in many compilers that several subroutines can be compiled without any control cards or other separators between decks. To produce such a system, it is usually necessary to have the compiler not read beyond the END card until it is prepared to process the next program. Note that in a monitored system, this is one of the interface considerations.

Program restarting. Some compilers have a standard restart procedure. Is this desirable? Perhaps in certain real-time situations it isn't practical. In free-standing systems, on the other hand, the ability to restart is quite useful. Having to reload your program and all of the library and run-time package from, say, paper tape or cards can be frustrating and/or infuriating, depending on the time of day.

Character set. In what character set (or sets) does the system function? BCD or EBCDIC? (Or something else?) In the case of a system that is supposed to handle more than one character set, how does it determine which one? Does it handle reading *and* punching correctly, both at compilation and execution times? What about magnetic tapes? These factors should be considered when you are writing the specification, not when you are getting ready to accept the completed software.

Unit record processing. What constitutes a record on the typewriter and on paper tape, in BCD and in binary? What is the effect, if any, of such characters as delete, carriage return, or backspace? What about gap (i.e., blank tape) on paper tape?

During BCD output, does the FORMAT statement control the physical length of records on magnetic tape? Between what limits? (Some magnetic tape units balk at one-character records.) What is the maximum size of a BCD record? What happens if you try to read or write more than this number? These questions are sometimes answered in the reference manual, but they may require specifying in the *monitor* as well as in the FORTRAN system. Otherwise a system can result that does not correctly punch fewer than 80 columns on a card, for example.

What is the format of a binary magnetic tape record, as produced by a binary WRITE statement? An improperly chosen format can triple backspacing time (and break tape units) and fail to handle a tape that contains both BCD and binary records. Also, should this form of I/O be usable on other media such as cards? If not, what happens if you try it?

Statement of intent

The kind of compiler specification we are considering will probably be a long document, and it is easy for the writer, as well as the reader, to get bogged down in details. Therefore, a helpful thing you can do, for yourself as well as your implementation

group, is to write a brief statement (say half a page to a page) telling what the compiler system is and is not intended to do.

For example, one could say of the SDS 9300 FORTRAN IV system that it is a three-pass compiler intended to operate on a 16K machine with at least three magnetic tapes. It is intended to run under the 9300 Monitor, with which it can co-exist in 16K. The compiler emphasizes richness of source language, lucid and elaborate diagnostics, and local optimization of object code. It does not do global optimization. The source language emphasizes compatibility with other FORTRAN languages, particularly FORTRAN II. This does not mean that every FORTRAN II program will run directly with no modification, but a good percentage of them will. The compiler is written interpretively, but the object code runs directly on the hardware.

A statement of intent not only makes the specification easier to understand, but also, during implementation, it clarifies what changes would or would not be in harmony with the intent of the spec.

The implementation

When software is produced by a group within your company, organizational problems can arise, but you *are* able to communicate with the implementation group as the software is developed. On the other hand, when a specification is to be implemented by an outside consultant, some special considerations apply.

Help for the vendor

A vendor producing software for you is in some ways like a customer. He is writing a program that must work on your equipment. Often his programs must be assembled on your assemblers, checked out with your utility programs and debugging aids, and interfaced with your monitor. One of your *customers* writing such a program has access to a hierarchy of sales support personnel to help him — to show him how to use the hardware and software, and to track down and get action on equipment that is down or bugs in the software. But seldom is such service extended to vendors. Without it, the vendor either wastes time trying to make do for himself, or (more commonly) he requests help from anyone he can find. The person he finds probably isn't expected (nor does he have time) to work on such things. It should not be the duty of the project leader of your monitor, for example, to show the vendor how to load his program, or to write a magnetic tape dump for him.

Acceptance tests

To begin with, the existence of a specification such as we have described will simplify the writing of acceptance tests. It's hard to write acceptance tests when you don't know what the system is supposed to do. Also, obtaining improvements is a laborious task when the vendor can see no obligation to provide them.

Even with adequate specifications, acceptance tests should not be static. There is a tendency to assume that acceptance tests can all be written in advance of the system delivery, and never improved or added to thereafter. This just isn't true. An acceptance test is like any other program; it has to be debugged. You set out to write a program that uses certain features in certain ways, but until you actually try the program, you don't know if it does what you intended. In a fairly standard language like FORTRAN, many features can be checked out on some other system (if you have access to one that works), but new features in the language, different precisions in the machine, and similar factors usually make complete checkout impossible.

Furthermore, it isn't reasonable to expect that complete acceptance tests can be constructed for any system entirely before its appearance. In theory, of course, you *could* construct a set of decks that tested every possible situation, but this would probably require a roomful of cards and more man-years than the system it's meant to test. In practice, you write as many tests as you can, based on the specifications, and you find the rest of the problems by observing the product. This means not only watching it function, but also (if possible) reading the listings. The latter is not always practical, but it serves several purposes. It may uncover weak points in the code and it helps determine by actual test the quality of the comments and flowcharts. After discovering an apparent error in the coding, you can construct a test program that demonstrates the error. Even if you don't read the listings, you can gain much insight by watching the system behave. Often a system's reaction to one situation leads you to suspect that it may not handle some other.

The arrangements for acceptance tests should be worked out in a way that acknowledges their nonstatic nature. Laying aside a specific period of, say, a month for running the acceptance tests is not a good idea. One bug can exist at the beginning of that period that makes it impossible to run 80% of the acceptance tests, and it may take more than a month to fix the bug. Defining a precise acceptance procedure is rather complicated, and differs for each system, but it should take into account the following:

- Testing may take quite a while; if so, it should

not be cut off arbitrarily.

- You should not have to provide all the acceptance tests in advance.
- The tests should be subject to change as they are debugged and improved.

This may sound harsh on the vendor, but it really shouldn't be. For one thing, if there's an adequate specification, he has little excuse for not meeting it. For another thing, why should software vendors get off so easily? The hardware vendor doesn't fare so well with *his* customers. They don't pay for machines that aren't working, and sometimes they go on not paying until they are satisfied that the system is acceptable. And they are judging the software too, of course. You ought to have the same right.

Check-points in the schedule

Meaningful project check-points are a particularly desirable feature in software implementation. Mere *delivery* of either software or documentation is not sufficient; there must be some verification of what is delivered. We have observed, for example, a check-point that was met by delivering an off-the-shelf manual that described, in glowing detail, an old system for some other machine.

CONCLUSION

It should be obvious from the preceding discussion that complete specifications for a software system are not trivial to produce. They require time and manpower. They may, nonetheless, be cheaper than dealing with the problems that arise when you don't have them.

However, it is not necessarily the case that *you* have to produce such a specification when the work is to be done by an outside consultant. If you know what to call for, you might expect the vendor to produce this document, subject to your approval. Or you might have one vendor write the specification, and another implement it.

One problem with receiving such a specification is that someone has to approve it, and this is far from easy. It is, unfortunately, not very hard to produce a specification that is both unreadable and incomplete, and the presence of the former trait greatly enhances the already monumental task of detecting the latter. This may mean that you ought to produce as much of the specification yourself as possible. It also means that before being accepted, the specification should receive the widest possible distribution (one person is liable to catch what someone else has missed); instead, just the opposite tendency is often observed.

Remember, no specification is absolutely airtight. A consultant who does not have your best interests at heart can probably come up with something unacceptable, yet still meeting the specification, if he tries hard enough. This can happen in-house too, of course, so it's hard to know whom you can trust. The combination of a proper specification and a person closely monitoring the work should help alleviate these problems.

One final pitfall has to be avoided: if you produce a specification as detailed as the one we've described, and then refuse to accept any changes to it, probably no one will bid on it. For example, you might specify a certain kind of diagnostics, while the vendor has written 47 compilers using a different (but equally good) technique. Since your intent is to come up with the most effective and appropriate system, listen to and study what the vendor has to say, and then come up with a specification that is acceptable to both parties. There is little to be gained in the long run by luring vendors into contracts whose magnitude they don't fully appreciate.

Furthermore, you should make provision to update the specification from time to time. You know this is going to happen. Hardware configurations change; customer requirements change; vendors are unable to complete certain items; etc. There should be someone whose job it is to keep the specification up to date, so that questions of the form "How is such-and-such currently defined?" can be answered without digging through random memos and letters. This is a rather clerical task and should not be given to a senior technical person, particularly since it can become a full-time job on a major software project. Also, when changes need to be made, the procedure should be as simple as possible for the technical people concerned. Often changes are not made, or are made but not documented, simply because there is too much red tape involved.

One facet of this paper needs to be re-emphasized. When we raise a question such as "Should the compiler produce an object listing?", we are not necessarily implying that it should. We are saying that you should know *in advance* whether it will or not, and specify accordingly. There may even be cases where you don't care one way or another about some feature (or it's not worth the time required to specify it). In that case, you should clearly *specify* that you don't care, so that neither you nor the vendor can get stung when you see what he has done. Unpleasant altercations can be avoided if you can say "That is not what we specified." or the vendor can say "You said you didn't care about this."

The important thing is to know what you're buying. You wouldn't dream of buying a piece of hard-

ware without knowing something about its components, size, temperature stability, response time, power consumption, external interfaces, ease of use, etc. But it is common practice to buy software without having the slightest idea what you'll be getting.

ACKNOWLEDGMENT

The authors wish to acknowledge the inspiration and invaluable assistance provided by Mr. Sol Zasloff and Mrs. Nancy Foy, both of SDS.

Observations on high–performance machines

by D. N. SENZIG
IBM Corporation
San Jose, California

INTRODUCTION

The high speed computer area seems to be dominated by a continued reduction in the price of computer switching circuits and the approach of these circuits to speeds at which the velocity of light becomes an important factor. Barring some unforeseen dramatic change in technology, the outlook for increased computational speed in the classical sequential machine organization becomes increasingly grim. Simultaneity, or parallelism, therefore, becomes more and more essential if computer performance is to continue to increase.

This paper discusses some limitations to the continued extension of the conventional look-ahead design. Two possible modifications of the conventional organization are introduced in a general way and discussed together with areas of application.

From an alternate point of view, this paper is a programmer/mathematician's proposal for an array processing machine. The ideas presented here are based on taking advantage of the inherent parallelism of the mathematics and/or program topology in addition to the local dependencies in the instruction mix exploitable by look-ahead. An effort is made to analyze the proposals in terms of cost and effectiveness. The discussion is limited to CPU organization, though it is recognized that this is but one of the potential bottlenecks in system performance.

Limitations to single processor performance

Representative high performance single processor machines are the IBM 7030 (STRETCH),[1] the Ferranti Atlas,[2] the CDC 6600,[3] and the IBM System/360 Model 91.[4] In these machines the control unit is responsible for executing certain operations or sequences of operations simultaneously. A machine of this type is designed to appear to the casual programmer as an ordinary sequential machine with the control unit having the responsibility for sorting out all possible dependencies in the instruction stream. This characteristic makes it necessary to restrict the design to detecting simultaneity on a relatively small local scale.

A programmer concerned with interrupt handling in a highly parallel look-ahead machine might find himself very much concerned with the fashion in which instructions are actually executed, as opposed to the sequence in which they are written. Division and memory protection errors illustrate the point. A division operation, even in a fast machine, can take many cycles. (When the unqualified term "cycle" is used, the reference is to the smallest time division in the control unit.) After the division operation has started a sophisticated look-ahead control unit can initiate independent instructions in the code following the divide. In the event an exception condition, such as exponent overflow, occurs, elaborate and costly recovery procedures are necessary to provide the system with sufficient information to effect recovery or precise information as to where the interrupt occurred. This example can be argued as unimportant, since exponent overflow is normally a catastrophic program failure and corrective action should be taken before the program is run again. The same sort of situation can conceivably occur in a paging operation where the pseudo address is generated from a series of computations possibly involving division and hence not be detected until a number of succeeding instructions have been executed. Here the interrupt is not due to a program bug but to a peculiarity of the instantaneous job mix. It is possible, of course, to legislate these problems away, but at the cost of a severe limitation on processor performance.

Another limit to processor performance that arises is due to circuit restrictions, even in the case where circuit cost is not a factor. With a given circuit fan in, for example, a lower bound on the time required to perform addition can be obtained.[5] An obvious way to obtain higher effective speed is by replicating functional units (adders,

multipliers, memories, etc.). But the same circuit restrictions also seem to put a limit on the number of instructions that can be decoded and initiated per unit time, and hence on the number of functional units that can be kept busy. A measure of the resulting circuit complexity can be had by noting that if U autonomous units are available and the instruction unit can initiate I instructions, then U^I combinations have to be detected by the instruction unit.

Array processing

The above considerations lead to the possibility of implementing more powerful instructions as a means of avoiding the instruction decoding problem. Instructions to take advantage of the inherent parallelism in the basic operations of linear algebra turn out to be a particularly useful place to start since these are the basis for most scientific calculation. Fox[6] states that "linear algebra is involved, wholly or in part, in 80% of all scientific calculations." The operations and techniques used here can, in all probability, be used equally well in such "non-scientific" calculations as payroll and inventory control. The scientific area is easier to discuss since the same well-known algorithms are used in a variety of applications.

The array operations are developed by generalizing the conventional fixed point, floating point and Boolean instructions to apply to an array of data (a vector) rather than a single operand (a scalar). An instruction contains three parameters which specify a vector: the address of a reference element in storage (the base address), the spacing between the elements in storage and the number of elements to be processed. Since the number of elements in the vector (order of the vector) is specified by the programmer, the normal register-oriented CPU design with a fixed number of registers is not attractive or feasible. Hence, all vector instructions (those that specify an array of operands) functionally operate memory to memory. The CPU would undoubtedly include high-speed registers, but the allocation would be done by the hardware. Techniques similar to those used in one-level-store machines[1,7] to automatically match a large capacity, slow memory to a smaller, faster memory (here an array of registers) are attractive and feasible.

A typical instruction would add the n numbers stored at locations a, a+S, a+2S, . . . , a+(n−1)S; to the n numbers stored in the consecutive locations, b, b+1, b+2, . . . , b+n−1. The n sums would then be returned to locations b, b+1, b+2, . . . , b+n−1. The instruction specifies the base addresses a and b, the spacing S, and the order n in addition to the operation, add. The two address format is chosen for illustration partly based on Amdahl's contention[8] that it is slightly more efficient than three address. Since at least one of the imput operands to an instruction is normally the output of an intermediate calculation, forcing that input operand and the output to occupy sequential locations is not a serious restriction in practice. Further, the two address format simplifies the design of the control unit and simulation indicated it tended to encourage efficient use of the one-level store.

As stated above, the vector instructions include the conventional Boolean, fixed point, and floating point instructions systematically extended to apply to each element of the array and a number of special selection operations. These latter operations are discussed below. In addition to the vector instructions, the conventional index register instructions which one would find in a conventional computer are also defined.

We have said nothing about how the instructions are executed. It should be obvious, however, that given sufficient hardware the n operations specified in the instructions could be executed simultaneously; indeed, from the programmer's point of view they are.

In a sense, array processing machines such as ILLIAC IV[9] have proposed this approach previously but require the user to match the array sizes, etc., of his problem to the physical configuration of the arithmetic units.

Array processing storage requirements

While the vector instructions can potentially reduce the instruction decode problem they require simultaneous access to $n(n >>)1$ words in storage to avoid introducing a new bottleneck.

In ILLIAC IV, simultaneous access is achieved by each arithmetic unit having its own storage unit. While this is perhaps the fastest and cheapest implementation and is suitable in many applications, it is awkward to rotate or shift arrays or to permute the elements of arrays that extend across multiple storage units. Simulation has indicated that to process general linear algebra codes, it is essential to be able to fetch matrix rows and columns with a minimum of interference or time-consuming operations to avoid interference. The technique proposed here was suggested in the paper "Computer Organization for Array Processing."[10]

If the standard interleaved memory of 2^J autonomous storage units is considered, it is obvious that 2^J sequential words can be accessed simultaneously. Recalling that the vector instructions specify the spacing, S, between words, and that the number of storage units is a power of 2, *any* odd value of S will result in 2^J words being in different units from which they can be fetched or stored simultaneously. (That all 2^J storage units will be accessed before a second request is made to a unit can be seen by noting that S odd is relatively prime to 2^J.) If a multidimensional array such as matrix is speci-

fied as containing an even number of elements in some coordinate, then a dummy component can be added to achieve an odd number. The control unit must be designed to accept an even value of S but in general some loss of performance is to be expected when S even is specified. Vector instructions with one operand a scalar (S=0) and the other a vector is a sufficiently common situation so that it should be detected and only one storage access performed. Thus, the conventional interleaving of autonomous storage units on the least significant bits of the address can be used to access arbitrary cross section of arrays.

By interpreting the n elements fetched as a new set of addresses rather than data, indirect addressing is possible. The n words fetched in the first access are sent back to the storage address register to bring out n new words. This addressing technique will, in general, lead to multiple requests from the same storage unit. This will result in less than maximum speed, but will be faster statistically than executing n separate instructions, one for each word to be read or written. The indirect access can be extremely useful in permutation operations such as table look-up, interpolation, curve fitting, or in selection where the given operation is performed on a subset of the elements in the array.

Restructuring operations

As a result of writing test codes, particularly from the area of numerical weather forecasting, it is clear that the effective use of array instructions depend on the ability to restructure a data array by extracting and reinserting certain elements or groups of elements. This is the basis for the handling of exceptional conditions in the array. This restructuring depends on selection of elements, which is a binary operation, that is, select or do not select. Rather than make this selection on the basis of geometric select registers or mode bits, it is proposed that the *compress* and *expand* operations defined by Iverson[11] be implemented.

The *compress* instruction is defined as follows: Given the output vector C, the order and components of C are obtained by suppressing from an input vector A each component for which the corresponding bit of the control vector B is zero. If B=(1,0,0,0,1,1) and A=(2,3,4,5,6,7), the operation results in the vector C=(2,6,7). The number of ones in B is equal to the order of the result. This latter value is returned in an index register.

The *expand* instruction is, in a sense, the inverse of compress. The expand operation is equivalent to choosing the successive components of the output vector A from the initial components of the input vector C or from a pseudo vector all of whose components are zero according to whether the successive components of B are 1 or 0. For example, if B=(1,0,0,0,1,1) and C=(2,6,7) then the result is A=(2,0,0,0,6,7). The order of the array specified in the instruction refers to the vector B. Denoting the sum of the ones in B by i, the i elements originally in C will be preserved and n—i components of the result are necessarily zero. In the case where a two address format is preferred the vectors A and B could occupy the same memory locations.

The restructuring operations are typically used in situations where data dependent branches occur. One can make a pass over a primary array, separating it into multiple sub-arrays by use of compress together with indirect addressing. One then does the necessary calculations on each sub-array separately with suppressed elements not participating in any sense. The expand operation then allows the sub-arrays to be reassembled into a new primary array for use in the next procedure.

Multiple control units

At the risk of some over-generalizations, the proposals above find considerable use in linear algebra based codes. In addition to a test world weather prediction (general circulation) code, a number of library routines, such as matrix multiply and inversion, linear programming, numerical quadrature, various over-relaxation schemes for partial differential equations, among others, were coded. These routines for the most part merely take advantage of the inherent parallelism of the algorithms. Performance would tend to be limited only by the bandwidth of the functional units employed or by the order of the arrays involved.

However, ordinary differential equations, the finding of polynomial roots, and Monte Carlo problems, do not exhibit this type of parallelism. Parallelism is inherent in the algorithms but the code applied to each data item or equation is different. The usefulness of the vector instructions also declines rapidly if one attempts to use them in calculations involving a great deal of branching conditional on intermediate results generated during a computation. Further, instruction execution statistics[12] imply that the functional units are going to be free an appreciable fraction of the time.

The above considerations lead one to consider the time sharing of the functional units among multiple control units. In particular, the current thinking is toward a system whose physically independent CPU's consist of a collection of instruction counters and instruction registers. These registers, the index registers, and possibly some additional programmable registers, appear to the user as physically independent and complete CPU's. Each CPU has access to, but time shares such units as memory (subject to protection restrictions), multiply/dividers, floating adder/subtracters, and instruction decoders. This mode of operation permits considerable

hardware reduction with little loss of performance. If one is concerned about precise interrupts, then the ability to rapidly switch tasks when an interrupt may be ambiguous permits the system to proceed. Thus, the functionally independent CPU's need physically consist of only the visible programmable registers (control unit). From a pragmatic point of view, the control units also do locally such relatively simple operations as index add, and Boolean operations, i.e., those operations that it would be faster to do locally than test the status of a shared unit. A block diagram of a possible system is shown in Figure 1. (The number of units of a given type is not intended to imply an optimum.)

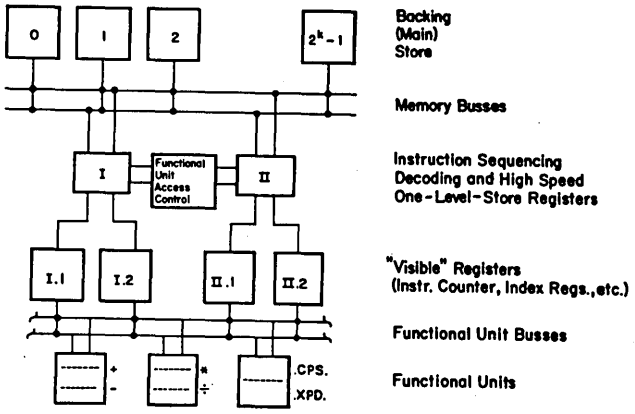


Figure 1—Block diagram of a possible system

Task synchronizing instructions

A number of papers have appeared proposing instructions that would be implemented to synchronize multiple instruction streams. Conway's[13] is perhaps the best known. Rather than add a new set of names to the list emphasizing another set of prejudices, we will use operations based on those defined in Operating System 360 (OS/360).[14] In OS/360 the operations are implemented as macro instructions but could be wired-in or microprogrammed if noticeable performance gains could be shown to exist.

The fundamental operation needed to control multiple CPU's is one that permits a storage location to be updated with no possibility of other processors, including I/O channels, accessing the subject location and obtaining out of date information. A programming solution to the problem has been proposed by Dykstra.[15] An instruction, Test and Set, is provided in the IBM System/360 and Univac 1108 machines for the same purpose. Given Test and Set, the other instructions described here could be constructed by the programmer. But again it appears reasonable to let the hardware assume a reasonable share of the burden.

In what follows, the term TASK is used for the work done by the execution of a section of code. This definition is necessary to permit the same section of code to be used by a number of tasks, perhaps from different jobs, perhaps simultaneously.

To effectively employ multiple CPU's a job must be able to create additional, potentially simultaneous tasks. This is done by ATTACH. The ATTACH instruction specifies a name for the Task, the location of the first instruction to execute, and a parameter list. Assuming a CPU (Control Unit) is available the supervisor assigns it to the Task initiated by the ATTACH instruction. If no CPU is available to begin execution the Task is queued. The Task queue is to be available to all CPU's. Hence the user need be concerned only with the Tasks he initiates, not with the processor which executes the Task.

A DETACH instruction causes the named Task (necessarily from the same job) to be removed from the system.

Where two or more Tasks may wish to use a facility, such as a section of code that is not re-entrable or an I/O device, a means of queuing the Task request is necessary. The ENQUEUE instruction causes the Task to be placed on a queue for the facility if the facility is busy. If the facility is not busy when the ENQUEUE is executed an indicator is set to indicate busy to succeeding requests and the Task proceeds. When finished with the facility the Task releases it by means of a DEQUEUE instruction.

A change priority instruction, CHAP, is provided which permits the Tasks of a given job to change their priorities relative to each other.

A Task may enter a waiting state implicitly by, for example, executing the ENQUEUE instruction. It may also enter the waiting state by executing the WAIT instruction. WAIT specifies that the Task should continue only when one or more events, such as an I/O operation or the completion of a procedure, have occurred.

The POST instruction is used by one Task to signal another that an event required to satisfy a WAIT instruction has occurred.

In addition to requiring dynamic storage allocation because of re-entrant coding, the amount of temporary or working storage required when the vector instructions are used is generally not known until array sizes are specified during execution. GETMAIN requests the allocation of additional main storage to the task. The FREEMAIN instruction releases main storage.

These proposals would permit fully simultaneous execution of general instruction sequences as specified by the user. The multiple control unit machine is here proposed mainly for the fast solution of a single problem. Time sharing, graceful degradation, and multi-program-

ming activities, which are important potential byproducts, are here considered to be of secondary importance. But it should be noted that there is much in common between the single job oriented multiple task organization described above and a supervisor whose "tasks" are multiple independent jobs. That is, to the supervisor the programmer specified jobs can be considered tasks that interact through their requests for common facilities. In either case the necessary protection features must be provided.

With the multiple control synchronizing operations a number of hand coded subroutines have been written and analyzed which merely take advantage of the inherent parallelism of routines such as non-linear ordinary differential equations, the Monte Carlo solution of diffusion equations, etc. Though it is a matter of opinion these routines turned out to be only very slightly more difficult to write than routines for serial computers of 7090-type organization, and considerably simpler than trying to optimize the codes for a look-ahead machine of STRETCH or 6600 type. In general, quite straightforward coding procedures here and in the linear algebra codes give excellent results in cases which represent large fractions of computer time.

The complete instruction set would then consist of the normal bookkeeping instructions, the vector instructions described above, plus the special instructions for synchronizing the asynchronous instruction streams.

Logically, the multiple instruction streams obviate the need for the vector instructions. However, where applicable the vector instructions permit the processing power of the system to be placed under control of a single control unit while the control of multiple instruction streams must be done through a supervisor which means appreciable overhead. The overhead seems to be especially bad in the linear algebra codes where the main computational loops tend to be short requiring frequent synchronization.

In simulation studies a vector instruction based organization also tended to be faster than a look-ahead or multiple control unit organization with the same arithmetic and instruction unit bandwidth. This occurred because the vector instructions naturally used the memory bandwidth to the maximum in accessing arrays while with the other organizations the tendency was to access elements from different arrays. This led to essentially random requests to the storage units and, hence, interference. Of course, the careful coding of the look-ahead and multiple control unit machines, in particular unwinding loops to contain two or more iterations, can reduce the interference to equal levels.

It should be emphasized that the multiple control unit synchronizing instructions and the vector instructions are to explicitly inform the system that possibilities for simultaneous execution exist. Whether these possibilities are taken advantage of would depend on the mix of tasks and hardware availability at run time.

Functional unit design considerations

If a machine is to process $n(n>>1)$ data points simultaneously as implied by the vector instructions and multiple control units, it is convenient to provide a physical arithmetic unit for each instruction counter. In addition to programming difficulties in fitting a problem whose array sizes vary from run to run (or dynamically during a run) to a fixed size arithmetic network, other problems can arise.

(1) To keep control to a minimum during the execution of the array instructions, all arithmetic operations should run in step and the time required should be data independent. Floating-point add, in particular, gives trouble because of its data-dependent shifts.

(2) Considering the memory organization, ability to simultaneously transfer 2^j words to n arithmetic units requires a full cross bar switch or restrictions on how much memory is available to a given processor.

(3) If the program cannot be put in array form, the system runs at the speed of the slow (presumably) arithmetic units.

(4) How the arithmetic unit network is to be segmented when multiple control units are simultaneously working on independent jobs is not clear. Providing the physical connections to partition the arithmetic units among the control units is not a problem. Rather, the problem occurs when attempting to code subroutines, etc., that will run reasonably well with a network that can be partitioned in a number of ways.

(5) The formation of the inner product of two vectors ($\Sigma_i a_i b_i$) is a fundamental operation of linear algebra. The formation of the sum requires an elaborate switching network within the processors.

Rather than work with a network of arithmetic units, a much smaller number of high speed pipelined units can be used. The pipeline operation can probably best be illustrated by the floating-point add operation. We assume two vectors A and B are to be added to produce a sum vector C. Assuming that the memory fetch part of the operation has been completed and the vectors are in registers in the CPU, the pipeline could work as follows. On cycle 1, the first pair of operands is transferred from the A and B registers to the floating point add unit and the radix points are aligned by shifting the number with the smaller exponent. On the second cycle the fraction portions of the numbers are added. Simultaneously, the

second pair of operands, A_2 and B_2 are transferred to the add unit and aligned. On the third cycle, three pairs of operands are involved, the sum of the first pair is normalized and transferred to the destination register, etc. The process continues as illustrated in Figure 2 until all operant pairs have been added and the sum placed in C. Multiply and divide are executed in a similar fashion. A snapshot of a pipelined operation would reveal many operands being processed simultaneously, but each at a different point in execution.

Cycle No.	Xfer to Adder Unit and Align Shift	Fraction Add	Normalize Shift and Return Sum
1	A_1, B_1		
2	A_2, B_2	A_1, B_1	
3	A_3, B_3	A_2, B_2	C_1
4	A_4, B_4	A_3, B_3	C_2
5	A_5, B_5	A_4, B_4	C_3
6	⋮		

Figure 2—Pipeline adder flow $C \leftarrow A + B$

Pipelining seems to solve the problems mentioned above quite well. Data dependent execution times are eliminated by making the functional units essentially multi-level combinatorial circuits. The memory busses need only be sufficiently wide to supply words at the rate required by the arithmetic units, rather than providing for simultaneous transfer. Non-vector problems will see high performance arithmetic unit time. While there may be conflicts over the use of the execution units, the programmer will not have to worry about segmenting a network. The add portion of the inner product can be done by gating the output of the added back to the input. Searching for the largest or smallest element in an array can be done by noting the sign of the output and saving one or the other of the two inputs.

It is convenient to envision the pipeline units as dedicated multiply/divide units, add/subtract, and restructuring units. Preliminary study indicates that when the units are specialized to perform a given function, speed and design simplicity result. Multiplier/divider and floating point adder/subtracter circuits that possess these characteristics are discussed by Wallace[16] and Anderson[17] respectively.

As was mentioner earlier, there is a limit to the performance a given functional unit can deliver. When higher performance is required, units can be replicated without increasing the instruction decode problem since the programmer has provided explicit information as to which operands can be processed simultaneously. The hardware is to have the responsibility as to which operand is to be processed by which functional unit. Also, if adjacent instructions use different sets of functional units the overlapped processing of operands from separate instructions is possible.

Programming systems

A compiler which would accept standard FORTRAN or PL/1 coding and analyze it for implicit simultaneity has been considered only briefly. Though many of the techniques currently used in compiler optimization are applicable, the coding of such a compiler presents many problems. Because of the easy with which many significant problems can be adapted to the multiple control unit machine it is this writer's belief that the design and coding of such a compiler presents a more useful study than studies that show the applicability of this or that algorithm to a machine with multiple control units, arithmetic units, etc. This is not to say that the extension of existing problem-oriented languages to include the explicit specification of array operations and potentially simultaneoussections of code should not be done. The skilled coder should certainly be provided with facilities to enable him to take advantage of special knowledge and work with maximum efficiency. The PL/1[18] language permits the specification and control of multiple tasks based on the multiple control unit synchronizing instructions described above.

Iverson has proposed a language[19] based on his notation where the array operations defined are consistent with mathematical logic. The richness of the symbol set proposed by Iverson probably precludes its direct implementation in production facilities where constraints on the character sets of keypunches and printers exist. FORTRAN IV got around the character set problem by enclosing letters with periods to form new operator symbols, e.g., .N E. for $\neq$. Using the compress and expand operations described above as an example, these would look like

C = B CPS A(,5)

or A(,5) = B XPD C

where B and C are vectors (defined as such in a declaration statement such as the FORTRAN dimension statement) and A(,5) represents all elements of the 5th column of the matrix A. Following Iverson's proposal, simultaneous displacement is meant when an array appears on both sides of the equal sign in an assignment statement. The standard arithmetic and logical operations are extended systematically to apply to each element of an array, and some special operations such as Compress, Expand, and Sum ($\Sigma_i x_i$) are required. From these operations the basic operations of linear algebra such as matrix inversion and eigensystem evaluation are to be programmed.

There are two reasons for defining array operations in this fashion rather than defining the operations to be the basic linear algebra operations. First, the algorithm one uses, especially in matrix inversion and eigensystem analysis is highly dependent on array properties such as condition number or sparseness. Second, representation of unrelated elements as arrays, especially as arrays of dimension greater than two, is very convenient, particularly for I/O to temporary backing store. In this case, the operators one would define are unclear.

Complexity and performance

Without specific performance goals and hardware costs and specifications the absolute costs of these proposals cannot be obtained. But some general observations on the relative complexity (which is presumably related to costs) of look-ahead machines and the proposals presented here are possible. If the two systems have the same maximum performance then they have the same I/O requirement, the same memory capacity and bandwidth, and the same arithmetic bandwidth. Thus, for roughly equal performance, the two systems differ only in control hardware. If, as stated above, the hardware complexity for the look-ahead machine control can be measured as U^{I_l} where U is the number of autonomous units and I_l is the number of instructions per cycle then, since in an R control unit system the individual control units must decode instructions relating to R-I additional control units, the complexity of the individual control units is given by $(U+R-1)^{I_m}$ where I_m is the number of instructions decoded per cycle by an individual control unit. Since R control units were assumed, the total control complexity is measured by $R(U+R-1)^{I_m}$. The maximum performance of the R control unit system is given by RI_m. This analysis ignores a number of things, not the least of which is the interference and interdependence of instructions.

In Figure 3 three designs are assumed with U=4. The system is assumed to consist of a memory, an adder/subtracter, and a mutiply/divider in addition to varying numbers of control units. Figure 3 indicates that for a system performance of up to one instruction per cycle the single control unit machine is the least complex way of obtaining a given performance. From the view of this approximate analysis this is reassuring since despite all the papers on parallel machines they are notable for their less than enthusiastic acceptance in the market. Further, the most complex machines currently being produced are the single instruction counter CDC 6600 and IBM Model 91. Logically, both these machines have a peak performance of one instruction per cycle. Figure 3 indicates that above a one instruction

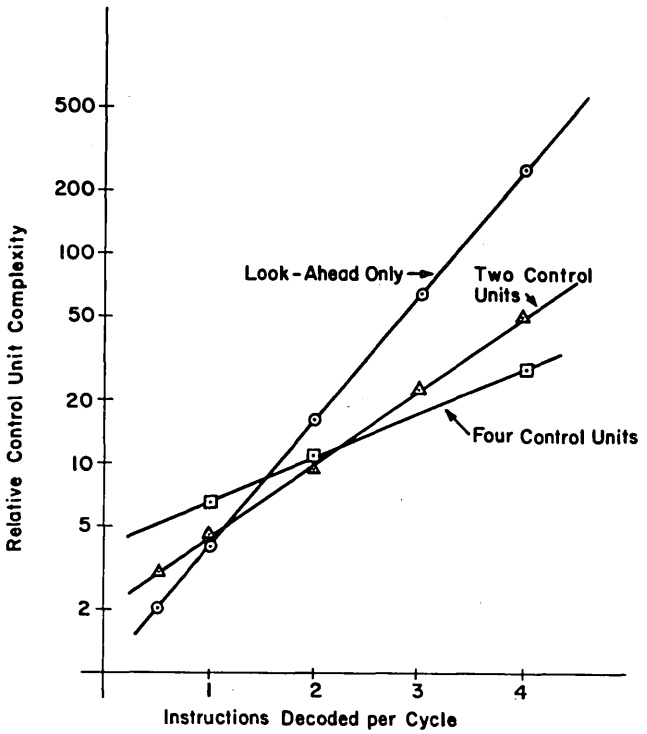


Figure 3—Control unit complexity for equal decode rate

per cycle rate the multiple control unit machines become competitive. The only machine discussed in the literature of greater logical performance than the CDC 6600 and IBM Model 91 is the multiple control unit, multiple arithmetic unit ILLIAC IV.

The curves in Figure 3 are of little interest above four instructions per cycle since we have but four autonomous units. In Figure 4 the number of autonomous units is chosen to be a function of system performance ($U=2R\,I_m$). This analysis again indicates that below one instruction per cycle the single control unit machine is the least complex way of obtaining a given performance. Above initiation rates of one instruction per cycle the programmer assistance required in multiple control units can reduce the complexity.

The above argument does not account for control units being idle because program logic dictates that execution must be sequential. It should be obvious that both the single control unit look-ahead dependent organization or the extensions discussed in this paper are going to present a highly problem-dependent performance. A "percent parallelism" curve due to Amdahl[20] can be drawn to illustrate this. If a machine with A functional units and sufficient instruction decoding runs at maximum rate, it could execute A instructions per cycle. Running at its slowest rate we assume it executes I instructions per cycle. If it executed 50% of the instructions serially at an average rate of I instructions per cycle and ran the remaining 50% at the maximum

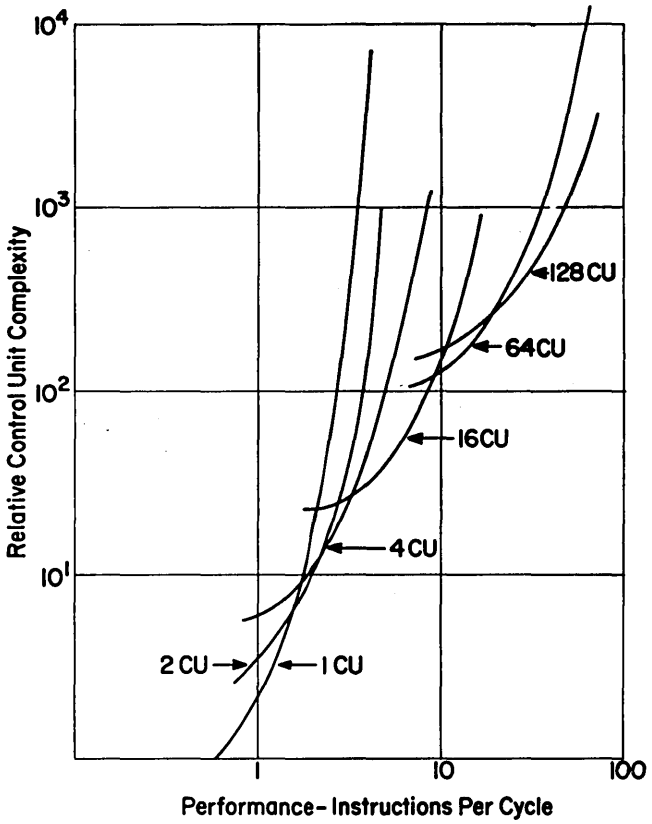


Figure 4—Control unit complexity and performance for various machine organizations

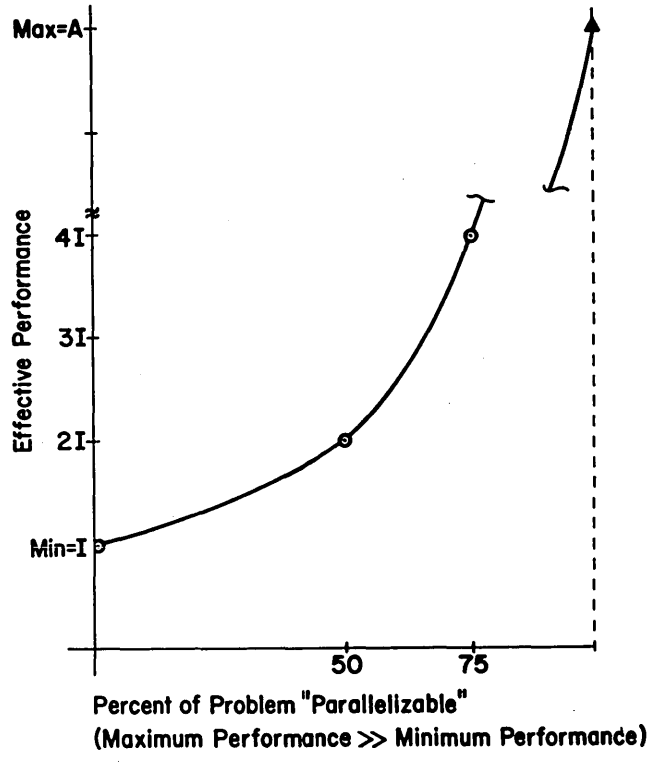


Figure 5—Problem dependent performance of parallel machines

rate machine performance would approach 2I instructions per cycle at best. If it ran 75% of the time at the maximum rate the performance would approach 4I instructions per cycle. The curve is plotted in Figure 5. The curve applies to any parallel system without regard for whether the multiple functional units are kept busy by look-ahead, multiple control units, vector instructions, or some combination. A given code will, of course, exhibit different amounts of parallelism to various organizations.

Figure 6 illustrates curves drawn for a number of different organizations. It is assumed in Figure 6 that the same maximum performance is feasible in all cases. We then see how performance can degrade when one relies on paralleling multiple units to achieve high performance.

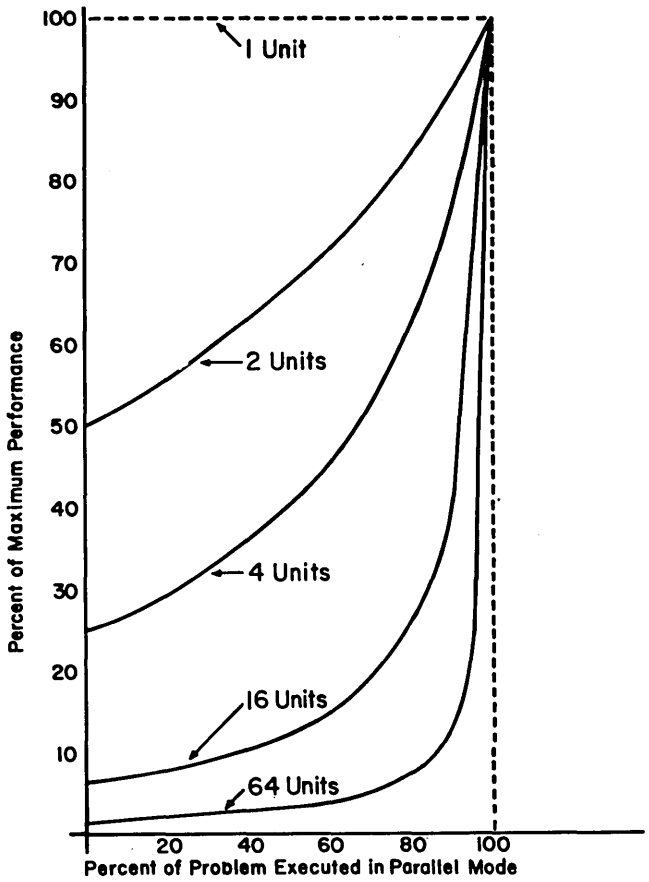


Figure 6—Problem dependent parallel machine performance

SUMMARY

The proposals presented here for extending the performance of machines differ from extensions of the look-ahead machine in that the look-ahead based organization hardware anticipates at every execution almost every action of the programmer to achieve local simultaneity. The proposals for vector instructions and multiple instruction counters rely on the programmer or

compiler to explicitly specify global simultaneity on the basis of an analysis done prior to execute time. For look-ahead to go much beyond the CDC 6600 and IBM Model 91, sympathetic programming of an increasingly gauche character will surely be required. The same care in specifying simultaneity based on mathmatical and program logic rather than hardware gives indications of working for simplicity in both programming and hardware systems. However, in requiring extreme parallelism to achieve performance we can expect problems where the lack of parallelism is going to present effective performance far below that claimed in the sales brochures.

REFERENCES

1 W BUCHOLTZ
Planning a computer system
McGraw-Hill Book Co Inc New York 1962

2 T KILBURN D B G EDWARDS M J LANIGAN F H SUMNER
One-level storage system
IEEE Transactions on Electronic Computers EC-11 223 1962

3 J E THORNTON
Parallel operations in the control data 6600
AFIPS Conference Proceedings 26 part II 33 1964

4 D W ANDERSON F J SPARACIO R M TOMASULO
Machine philosophy and instruction handling
IBM Journal of Research and Development 11 8 1967

5 S WINOGRAD
On the time required to perform addition
J.A.C.M. 12 277 1965

6 L FOX
An introduction to numerical linear algebra
Oxford University Press New York 1965

7 D H GIBSON
Considerations in block-oriented systems design
AFIPS Conference Proceedings 30 75 1967

8 G M AMDAHL
The structure of system/360 part III—processing unit design considerations
IBM Systems Journal 3 144 1964

9 D L SLOTNICK
Unconventional systems
AFIPS Conference Proceedings 30 477 1967

10 D N SENZIG R V SMITH
Computer organization for array processing
AFIPS Conference Proceedings 27 117 1965

11 K E IVERSON
A programming language
John Wiley and Son Inc New York 1962

12 R A ARBUCKLE
Computer analysis and thruput evaluation
Computers and Automation 15 no 1 12 1966

13 M CONWAY
A multiprocessor system design
AFIPS Conference Proceedings 24 139 1963

14 B I WITT
The functional structure of OS/360, part II—job and task management
IBM Systems Journal 5 12 1966

15 E W DYKSTRA
Solution of a problem in concurrent programming control
C ACM 8 569 1965

16 C S WALLACE
A suggestion for a fast multiplier
IEEE Transactions on Electronic Computers EC-13 14 1964

17 S F ANDERSON J G EARLE R E GOLDSCHMIDT D M POWERS
Floating point execution unit
IBM Journal of Research and Development 11 34 1967

18 *PL/1 language specifications*
Chap 6 Dynamic Program Structure IBM Form C28-6571-4 1966

19 K E IVERSON
Elementary functions
Science Research Associates Inc 1966

20 G M AMDAHL
Validity of the single processor approach to achieving large scale computing capabilities
AFIPS Conference Proceedings 30 483 1967

The Greenblatt chess program*

by *RICHARD D. GREENBLATT,*

DONALD E. EASTLAKE, III,

and

STEPHEN D. CROCKER

Massachussetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

Since mid-November 1966 a chess program has been under development at the Artificial Intelligence Laboratory of Project MAC at M.I.T. This paper describes the state of the program as of August 1967 and gives some of the details of the heuristics and algorithms employed.

Development of the program

The first step we took was to produce a simulated chess set, whereby the computer would display the current board and accept moves in standard chess notation through a teletype. Routines to evaluate the board, generate legal moves, and perform a minimax search of a game tree were quickly added, and with further development the program played in its first tournament in February of 1967. It played in local tournaments again in March, April and May. The improvement it has shown is due to additional programming and debugging, not learning.

Table 1 summarizes the program's performance in tournaments. For comparison, the mean of all U. S. tournament players is about 1800, while the mean of all chess players is in the 800 to 1000 range. The program wins about 80% of its games against non-tournament players.

Table 1

	Won	Lost	Drew	Rating	Performance Rating
Feb	0	4	1	1243	1243
Mar	1	4	0	1330	1360
Apr	2	0	2	1450	1640
May	0	4	0	1400	(weakest opponent was 1680)

The program is an honorary member of the United States Chess Federation and the Massachusetts Chess Association, under the name Mac Hack Six. In the April amateur (non master) tournament the program won the class D trophy.

A short history of chess playing programs

The first important paper dealing with methods for programming chess playing programs was written by Shannon in 1949 (1). In his paper the concept of minimax tree search is used. In 1950, Turing described a hand simulation of a chess program (2). In Turing's paper the concept of a dead position is introduced. A dead position is one in which neither side can immediately gain by making a capture. These papers and two programs known as the Los Alamos program and the Bernstein program (8) are described in a paper by Newell, Shaw and Simon (3). Their paper describes a chess program which deviates from the analysis done in the previous programs in that it employs explicit plans and goals in making its moves. A more recent program in the Newell, Shaw and Simon tradition is the MATER program of Simon and Baylor (4). This program, however, deals only with mating combinations of a few moves. The program which is most similar to our program is described in a Bachelor's thesis by Alan Kotok (5). A variant

*The program was written primarily by the first author who was assisted by the second author. Work reported herein was supported in part by Project MAC, and M.I.T. research program sponsored by the Advanced Research Project Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

of Kotok's program was used by John Mccarthy in a chess match with a Russian program (6). It is fair to say that our program is stronger than any of these programs in across the board play.

Approach and environment

The approach we have taken in writing the chess program has been quite pragmatic. We did not pretend to be writing a general problem solving system, but addressed ourselves directly to the problems of chess. The goal of being able to play complete games under tournament conditions has meant that most of the effort so far has gone into building an efficient and effective tactical base. Therefore, consideration of learning mechanisms, strategic planning mechanisms and special case treatment of opening and end play were forestalled. Book openings were recently added, although it turned out that the computer played much better in the openings without them than was expected.

The environment in which this program has been developed is, we feel, more advantageous than for any previous chess program. The machine used is the Digital Equipment Corporation PDP-6 in the Artificial Intelligence Laboratory of Project MAC. This machine is equipped with a 256K Fabritek memory, a DEC 340 graphic display, a model 35 teletype, a line printer, and four Dectape drives.

The machine was originally used on-line by one person at a time and the teletype and graphic display provided a high degree of interaction between the user and the program.

The software provides for the editing, assembling and debugging of programs and makes full use of the interactive facilities. The mass memory and a time sharing system were added after most of the initial work on the program was done.

The mass memory has proved very useful in later versions of the program, but it should be noted that at the time of the first two tournaments the machine had only a 16K memory.

The program was written entirely in MIDAS, a PDP-6 macro assembly language (7). MIDAS was chosen for this program because of the ease of constructing and debugging in it the complex data and control manipulations involved in writing a high performance chess program. Large economies of time and memory are also effected by writing in assembly language. The order code of the PDP-6 computer is exceptionally well suited to assembly language coding.

The program has been edited and reassembled over 200 times and has played several hundred complete games; consequently, those portions of the code which have been in use for a while are extremely reliable and the program's performance has yielded many ideas for improvement.

Debugging aids

The chess program contains several powerful interaction debugging aids. These are briefly listed below:

1) scope display of the board and game history
2) acceptance of standard chess notation input (e.g., P-K4)
3) scope display of evaluation at any selected node in the game tree
4) tracing of specific move in plausible move generator, displaying all factors that went into plausibility and a comment about each. (e.g., 10 points for unblocking the white queen bishop so that it now attacks QN5)
5) printed record of plausibility of all moves at top level and main variation from each top level move investigated
6) statistics on how long the computation took, how many plausible move generations, feedovers and static evaluations occurred, etc. (These terms are desicribed below.)

An outline of the program

We begin this section with a definition of some of the important chess terms and then describe the major components and the flow of control.

Chess terms

Ply—one play by one side. Two plies equal one complete move.

Pinned—a piece is pinned if moving it exposes (discovers) an attack on another piece, rendering that piece *en prise* (see below). If an attack on the king is thereby discovered, the original move is illegal.

Safe move—a legal move for a piece that does not render it immediately *en prise*.

Trapped—a piece is trapped if it has no safe moves.

Isolated, backward, doubled, tripled—various pawn structure defects. (See section on static board evaluator for further discussion.)

Development value—refers to a piece's range over the board (number of squares and importance of those squares) in a particular position.

Principal variation—the sequence of moves the computer thinks most likely in a position.

Game tree—the set of all positions considered by the program in a search, visualized in the form of a tree. This tree is diagrammed with the ancestor positions near the top of the page.

Game tree node—a node in the game tree represents a position. The line leading to the node represents the move which lead to that position. The lines down

from the node represent moves leading to successor positions.

En prise

A piece is *en prise* when it is under attack and is inadequately defended. An example is a knight under attack by a pawn and defended by a pawn (or any other piece). Clearly it is in the opponent's interest to take the knight even though he would lose the pawn. A more complex case is where a knight is under attack by a bishop and rook and defended by a pawn. In this case, it is the existence of a second attacker (the rook) which makes the knight *en prise*. The situation is further complicated when some of the attackers or defenders are pinned. Often a complete check for whether a piece is *en prise* can be quite complex, so in the program only partial checks are made at various stages. A typical determination is made by considering the value of the piece attacked, the number of attackers, the number of defenders and the values of the least valuable attacker and defender. *En prise* checks are made to determine whether or not the board is stable (in a dead state) and are also made at several places in the plausible move generator.

Description of a simplified minimax search

The program is organized around a minimax search of a game tree. The branches of the tree correspond to alternative moves and the nodes correspond to positions. Beginning with the actual position in which it is the machine's turn to move, a routine known as the plausible move generator lists each legal move and assigns a plausibility value to each move. The moves are then ordered according to their plausibility score and a subset of the moves is selected for further consideration. The first move of this subset is then postulated and the resulting position calculated. This process is repeated recursively until a certain depth is reached, at which point the position is evaluated using another routine known as the position evaluator. The position evaluator makes use of a function called the static board evaluator to compute a numerical value for the position. This numerical value has the significance that a positive value represents an advantage for white (the larger the number, the greater the advantage), a negative number represents an advantage for black, and zero represents an even game.

After a position is evaluated, the value is returned to the level above and it becomes the "best value so far" for that position. Each other move of the subset selected by the plausible move generator is treated in the same manner, and when a value is obtained for the move, the value is compared to the best value found so far. If the value associated with the move just considered is better for the side to move than the best value so far, the new move is remembered and its value becomes the new best value so far. "Better" is synonomous with "algebraically greater" if white is the side to move and "algebraically less" if black is the side to move. If two moves lead to the same value, it is presumed that the first is slightly better because it received a higher plausibility score. After all of the selected moves at a position have been considered, the best value so far and the move associated with that value are returned to the level above. The process is continued until a value for the actual current position is determined. The sequence of moves which are the best moves is called the principal variation.

Since it not feasible to consider either all moves at any level or an indefinite number of levels, some severe constraints are placed on the search. The basic search (just described) starts from the current game position and proceeds a fixed number of plies. A position evaluator is applied to each of the end positions of the basic search tree. This routine tests a condition known as the feedover condition (see below) of the position. If this condition is true, then the plausible move generator is reapplied (up to certain limits) and the position evaluator called at the resulting nodes. If the feedover condition is false, a value for the position is developed by calling the static board evaluator and by exploring all plausible captures. Plausible captures are generated in a manner similar to regular plausible moves, but they must appear to lead to relative gain of material, either through an actual capture or a pawn promotion. Positions resulting from plausible captures are turned over to the position evaluator. The program will explore sequences of favorable captures or pawn promotions without a depth or width limit. This is necessary because otherwise pieces might be left *en prise* and this would result in blunders of the first magnitude.

Should the program at any depth reach a checkmate, stalemate, or draw by repetition of the position, it will immediately return to the previous level with an appropriate mate or draw value. Also the alpha-beta algorithm may provide an exit at any level in the tree except the topmost level.

The details of the static board evaluator, the plausible move generator, the feedover conditions and the determination of the width of the search are all given In the next section. The alpha-beta algorithm is described in a later section.

The plausible move generator has three basic goals.

The plausible move generation

1) To select a subset of legal moves for inclusion in the move tree.

2) To order these moves so as to optimize the advantage the program receives from the alpha-beta tree-pruning algorithm.

3) To calculate the positional and developmental values that will decide the program's move if several moves lead to the same static value.

The analysis done in the plausible move generator is done on a per move basis rather than a per position basis—that is, for example, "this move is bad because it blocks my bishop" rather than "the position resulting after this move is bad because the bishop is blocked." To determine the latter fact starting just with the board position would require considerably more processing and analysis of irrelevent details.

Numerous heuristics are available for the plausible move generator. As is frequently the case with heuristics, they may not be valid in particular situations, therefore a program organization is required which allows for the interaction of the heuristics to determine which of them most nearly applies in the current situation.

Very generally speaking, two types of heurstic interaction are used in the chess program. One type of interaction involves enumeration of all combinations of facts. Such an enumeraion leads to the familiar tree structure with the nodes of the tree corresponding to subdecisions. Each node is dependent upon only one fact. The size of this tree grows exponentially with the number of facts involved, severely limiting the usefulness of this technique.

The second type of interaction uses weighted sums. A value is assigned to each fact proportional to its average importance, and each move is scored as the sum of the weights of the attributes which apply to the move. In the simplest case, the move with the highest score is chosen. The complexity of this process grows linearly with the number of facts; not exponentially. Also there is opportunity for a large number of small factors to add up and sway the final decision in a way hard to achieve with the enumerative process. While it is true that any linear weighting process can be simulated by an appropriate enumerative process, for large numbers of facts the size of the enumerative process becomes absolutely unmanageable. So for practical purposes the techniques are distinct. Linear weighting methods have been used before in game playing programs; nevertheless they have a weakness in that they basically fail to take into account the relationships that may exist between the facts. To put it another way, the importance of a fact may vary depending on the position. Non-linear techniques have been proposed to solve this problem, but chess is a game where the relationships are so complicated and numerous that it is unlikely much additional headway could be made by making the weighting nonlinear.

The solution incorporated in the current chess program is a nested combination of the two methods. The top level decision process is enumerative; that is a game tree is searched. However, selection of moves for the game tree is controlled by a weighted decision process, the plausible move generator. Many of the "facts" going into the plausible move score are themselves enumeratively determined using such criteria as whether the move is a capture or not, whether various pieces are *en prise* or not, etc. These predicates (or in some cases weights) are themselves decisions which are made by enumerative or weighted sum decision processes and so forth. The net result is that the program is frequently able to grasp the effect of particular features of the position that make some otherwise insignificant factor more important.

Details of the major components

The major reason for the quality of the program's play is that considerable chess knowledge has been programmed in. In this section much of the detail is presented. To some extent, these details are volatile, so what follows is more representative than definitive.

The plausible move generator

About 50 identifiable heuristics are used in computing the plausibility. Many, though, apply only in special cases such as captures, moves with certain pieces, or certain stages of the game.

Each square is assigned a importance during each plausible move computation, corresponding roughly to the estimated worth of having an additional piece bearing on the square or the cost of taking away a piece presently bearing on the square. The principal criteria used for assigning these values include the closeness of the square to the center of the board, its proximity to the opponent's king, and its occupation by one of our pieces which is *en prise*. Small values are given for occupation of the square by one of our pieces and for its closeness to opponent's side of the board.

The current developmental value of a piece is the sum of the values of all the squares it attacks (can move to in one move) plus values accumulated for actual attacks on enemy pieces. The new developmental value is similarily computed assuming the piece is in its proposed new location. The difference between these is used as a factor in the plausibility, encouraging developing moves and discouraging apositional moves. Gains or losses in development resulting from blocking or unblocking the opponent's or our pieces are also considered in the developmental value. Of course,

putting opponent's pieces *en prise* is plausible. Furthermore, factors are added to encourage certain types of attacks on probable weak spots (weak pawns, pinned pieces, pieces defending other pieces, etc.). When a capture is made, the capturing move receives the developmental value of the piece captured. Some very specialized heuristics also are employed, such as, "it is bad to move pieces in front of center pawns on their original squares, thereby tending to block your own center."

Several weaknesses were noticed in the early play of the program and measures were taken to eliminate them. For example, sometimes an apositional move would receive a high value because it was an attacking move. If this leads to gain, all is well and good; but if the opponent can simply move away then the move is a pointless waste of time. So, moves are scored separately on their positionality and if this is bad these moves are rejected if there is some other move which leads to an equal terminal score.

Evaluation of the board

The value of the board is given by

$$S = B + R + P + K + C, \text{ where}$$

B is a material balance term,
R is a piece ratio change term,
P is a pawn structure term,
K is a king safety term, and
C is a center control term.

The material balance term makes use of the evaluation shown in table 2.

Table 2

Piece	Value	Value Relative to Pawn
Pawn	128	1.
Knight	416	3.25
Bishop	448	3.50
Rook	640	5
Queen	1248	9.75
King	1536	12

The value of B is the sum of the values of the white pieces on the board minus the sum of the values of the black pieces on the board.

The piece ratio change term is aimed at promoting even or near even trades when ahead and avoiding them when behind. The ratio of white pieces to black pieces at the current node is compared to that ratio at the top of the tree. If the side to move is three pawns ahead, for example, a trade of a bishop for a knight will receive a positive piece ratio term.

$R = \{N/(T\text{-}1)\}*1/8*M$, where

N is the ratio of white material to black material at the node being evaluated,
T is the ratio of white material to black material at the top node of the tree, and
M is the material for one side at the beginning of the game.

The ratios are evaluated using the table above, except that the king is valued at 1 instead of 1536.

It has been pointed out that the piece ratio change is slightly asymmetric with respect to color, but this is of little consequence since this term only has effect when one side is very significantly ahead.

The pawn structure term depends upon four subterms, which score positively for each of the following: tripling up of opponent's pawns (doubling only if isolated), the isolation of opponent's pawns, our own passed pawns, and the opponent's backward pawns. Backward pawns are considered weaker if they occur on an open file or if the opponent has rooks or queens on the board.

A pawn is isolated if there are no friendly pawns on an adjacent file.

A pawn is passed if there are no enemy pawns in front of it in the same file or an adjacent file.

A pawn is backward according to the following criteria:

If it is defended by a pawn, it is not backward.

If it can be defended by a pawn in one move, (assuming moves through friendly pieces are permitted), it is not backward unless it is on the second rank and the only pawn move which would defend it is a double advance which would then subject it to *en passant* capture.

If there is a defending pawn move blocked by an enemy piece, if the pawn is blocked, the pawn is backward. If an adjacent pawn is blocked, the pawn is not backward.

Otherwise, if there are friendly pawns in adjacent files such that the pawn would become defended if advanced far enough, the pawn is backward. Otherwise, the pawn is not backward (i.e., it's probably isolated).

The king safety term applies only if queens are on the board. The king safety term (K) is eight times the rank of the black king minus eight times the rank of the white king.

The center control term (C) is +1 if there is at least one white pawn in the center four squares and no black pawn, —1 if there is at least one black pawn in the center four squares and no white pawn, and zero otherwise.

Feedover conditions

The feedover condition is true if:

1) the side to move has a piece *en prise* and one of the following:
   A) the side to move is in check.

B) the *en prise piece* is trapped or pinned.

2) The side to move has two or more pieces *en prise*.

3) Both sides have exactly one piece *en prise* and the piece of the side not to move is trapped or pinned, while the piece of the side to move is not.

The reasoning behind the first two of these conditions is that while the side to move could undoubtedly save a piece that was simply *en prise* he might not be able to save two pieces, both *en prise,* or one if it is trapped or pinned or if the side to move is also constrained to escape a check. Thus the side to move is forced to try his plausible moves and give the opponent an opportunity to try to capture the *en prise* material.

The reasoning behind the third condition is that the side to move may be able to save his piece instead of capturing the opponent's piece. Then the opponent will try to save his piece, which he may not be able to do since it is trapped or pinned.

The width of the search

Like the depth, the number of moves considered at each level is a constant tempered by some heuristics. The constants (a different one for each level) are usually all 6 for normal play, and are increased to 15, 15, 9, 9, 7 for tournament play, which means that the basic width at the top two levels is 15, while the basic width at levels three and four is 9, and the width is 7 for all succeeding levels.

The heuristics involved all have the effect of extending the width beyond the basic setting, so the only way that the program can fail to consider the indicated number of moves is either that the requisite number of moves simply do not exist or the tree-pruning algorithm provides an exit from the current level.

The heuristics are:

1) All safe checks are investigated.

2) At the first or second level, all captures are investigated.

3) An attempt is made to investigate moves of a certain minimum number of distinct pieces. This minimum is either half the basic width or the number of pieces with safe moves, whichever is less. This heuristic covers the case where all the moves of a single piece are highly plausible (say the queen, because it's *en prise*) and the rest of the board is not looked at.

4) Moves which lead to mate against the side to move are ignored and not tallied against the basic width. This guarantees that when a principal variation shows a mate, that mate is forced.

Additional features

Two algorithms for speeding up the search and three heuristic components for improving the reliability of the search comprise this section. The algorithms do not affect the quality of the program's play.

The alpha-beta tree-pruning algorithm

The alpha-beta algorithm (sometimes misnomered heuristic) has been a standard component of every modern game playing program. It was apparently first used by Newell, Simon, and Shaw (3).

In the search as described above, a move is discarded if it leads to a value which is worse for the side to move than some already considered move. If we look, however, at two levels of the tree, say moves by white, followed by replies by black, we notice the following: As moves by black are being explored, the value which is going to be returned back up to the white level (black's best value so far) cannot be getting any better for white and may be getting worse and worse. If a white move has already been evaluated, it is possible to check a black move not only to see if it is worse for black than some alternative, but also to see if it is so good for black that white would never make the move leading to that choice for black, or in other words, whether the move is "too good" for black. If the move is "too good," it is useless to consider any more moves for black from that position and the white move leading to that position may be discarded immediately. Thus, only one refutation is required to a proposed move and once it is found further search may be discontinued. The probability of alpha-beta cutoffs is increased by the fact that moves are investigated in order of decreasing plausibility, and a move is refued if it is equally good as the best so far at the previous level.

Such a consideration leads to a tremendous speedup of the search, especially if what turns out to be the best move at each position is considered first. One of the attributes of the plausible move generator is that it usually assigns the highest plausibility score to the best move, so almost maximal advantages is gained. (Rough calculation shows that the workload of the search is reduced by a factor of about one hundred.)

The name "alpha-beta" is derived from the fact that in the classic implementation of the algorithm, two recursive variables are kept: alpha, the best value so far for white, and beta, the best value so far for black.

Hash coding

One obvious way to speed up the searching process is to avoid considering the same position twice (as could happen through a transposition of moves). To

this end, the program incorporates a hash table into which an entry is made for each position considered. The entry records not only the results of the search but also a measure of how deep the search was which yielded the value. If the position is reached again, and the search in progress will not penetrate any deeper than the stored entry, then the results are immediately obtaind from the hash table. Due to the tree pruning algorithm, it is not always known exactly what the value of a node is, but only that it is greater or less than a certain value. Provision is made for storing this information in the hash table. On retrieval, the value is compared with alpha or beta (the tree prune variables) and a determination is made if further investigation is needed. Presently, the program uses a hash table of 32,000 entries with two machine registers per entry. An additional bonus of the hash table feature is that it enables the program to detect draws by repetition conveniently.

Modifications to the value returned by the search

If two moves are found by the search to lead to the same static value, the move which has the higher plausibility score is preferred. However, in some situations, this move is not the most desirable one to make. In order to take such cases into account, two types of small modifications may be made to the value returned from lower levels in the process of move tree searching.

The first modification subtracts a few points if the current move being investigated was marked as being developmentally poor by the plausible move generator.

The second type of modification occurs only if the principal variation that is returned is two or more plies long. If so, and it is found that the same piece was moved two plies down as is being moved in the current move, various small amounts are subtracted, depending on whether the piece is moved back to the square it came from or took two moves to accomplish a translation possible in one legal move or the position occurs during the first eight moves of the game (moves which are almost always devoted to rapid development). This second type of modification was introduced to give the program some sense of tempo and to counter its early tendency to make senseless attacking moves that were easily forced back.

Secondary search

A feature called secondary search was recently introduced. This was done in an attempt to obtain improved search depth at low cost. By increasing the depth of the search one can prevent the program from walking into traps which would not be recognized with a search conducted up to the normal depth. Moreover, one can discourage the tendency of the program to make delaying moves which force inevitable losses to occur beyond its normal lookahead. A secondary search is employed when the normal search results in a new candidate for the best move at the top level. What is done is to move down the principal variation for that move as far as this variation was computed by the plausible move generator, and then to conduct an additional search. The depth of this search is usually limited to two plies, although capture and feedover conditions can increase this number. The value produced by the secondary search is then used in place of the value first found for the principal variation if it is worse for the side to move.

This feature seems to improve the program's evaluation of many moves even though it is somewhat probabilistic in nature, since it looks at only a small subset of the positions that may be reached if the particular top level move is made. It seems to cause greatest improvement at tournament width settings when the principal variation is more reliable.

Book openings

The program incorporates a table of opening positions and selected replies. This "book" was compiled by two M.I.T. students, Larry Kaufman, a chess master and the top rated U. S. Junior player, and Alan Baisley, a chess expert.

The lines in the book have been selected to suit the computer's "style." The book contains over 5000 moves; however, actual games rarely follow book for more than approximately 10 plies. The book aids most the computer in avoiding "book traps" when playing against experienced players.

SUMMARY

Tournament play

The computer enters the tournament under the same rules as a human contestant. Moves are transmitted from the tournament site directly into the PDP-6 by teletype. A human operator is at the tournament who observes the opponent's move, types it in using standard chess notation, receives the machine's reply, plays it on the board and operates the clock. Of the two hours allotted to the machine for making the first fifty moves, about 7 minutes are normally lost in these operations.

The machine never offers a draw, but if the opponent offers one, the operator types in "draw?". The machine replies either "accept" or "decline." If the machine becomes hopelessly lost, human operators resign for it.

Results

The program is estimated to have played in excess

of 300 games in over the board competition with human players. It has played 18 tournament games. We will quote several tournament games. These games were played under rules calling for a minimum of 50 moves in two hours or an average of 2.4 minutes per move. Actually, the program played most of its recent games at about twice that rate. A single plausible move generation takes about 80 milliseconds for a typical position. In the early tournaments the computer did not keep track of the time used for each move, although this information is included with the game from a later tournament. The time quoted is actual computer time and does not include the operator overhead. This later tournament also saw the introduction of the book opening feature, which is not present in any of the other games quoted. (To convert the times given to machine operations, multiply by the PDP-6's approximate speed of 200,000 operations per second.)

Computer Tournament Chess Games

Tournament 1 the Winter Amateur Tournament of the Massachusetts State Chess Association Jan 21-22 1967

First Tournament Game Played By a Computer
White—rating 2190 Black Mac Hack VI

1	P-KN3	P-K4
2	N-KB3	P-K5
3	N-Q4	B-QB4
4	N-QN3	B-QN3
5	B-KN2	N-KB3
6	P-QB4	P-Q3
7	N-QB3	B-K3
8	P-Q3	PXP
9	BXP	N-Q2
10	PXP	R-QN1
11	B-KN2	O-O
12	O-O	B-KN5
13	Q-QB2	R-K1
14	P-Q4	P-QB4
15	B-K3	PXP
16	NXP	N-K4
17	P-KR3	B-Q2
18	P-QN3	B-QB4
19	QR-Q1	Q-QB1
20	K-KR2	N-KN3
21	B-KN5	R-K4
22	BXN	PXB
23	N-K4	P-KB4
24	N-KB6ch	K-KN2
25	NXB	QXN
26	N-QB6	QR-K1
27	NXR	RXN
28	Q-QB3	P-KB3
29	R-Q3	R-K7
30	R-Q2	RXR
31	QXR	N-K4
32	R-Q1	Q-QB2
33	B-Q5	K-KN3
34	P-QN4	B-QN3
35	Q-QB2	N-QB3
36	B-K6	N-Q5
37	RXN	BXR
38	QXPch	K-KN2
39	Q-KN4	K-KR3
40	QXB	Q-K2
41	Q-R4ch	K-KN3
42	B-KB5	K-KN2
43	QXRPch	K-KB1
44	Q-QR8ch	K-KB2
45	Q-QR8	Q-QB2
46	Q-Q5	K-N2
47	K-N2	Q-K2
48	P-KR4	K-R3
49	P-N4	K-N2
50	P-R5	Q-K7
51	P-R6	K-KB1
52	P-R7	QXKBP
53	KXQ	K-K2
54	P-R8=Q	P-QR3
55	Q-K6 MATE	

First Non-Loss By Computer in Tournament Play Game 3 Tournament 1
White—1410 Black—Mac Hack VI

1	P-K4	P-K4
2	N-KB3	N-QB3
3	B-B4	N-KB3
4	N-N5	P-Q4
5	PXP	N-QR4
6	B-N5ch	P-B3
7	PXP	PXP
8	Q-B3	Q-Q4
9	QXQ	NXQ
10	B-K2	B-KB4
11	P-Q3	B-QN5ch
12	B-Q2	BXB
13	NXB	O-O
14	P-QR3	P-KB3
15	KN-B3	QR-QN1
16	P-QN4	N-QN2
17	O-O	N-QB6
18	KR-K1	NXB
19	RXN	N-Q3
20	N-K4	NXN
21	PXN	B-K3
22	R-Q1	B-QB5
23	R/K2-Q2	R-QN2

24	R-Q8	RXR
25	RXRch	K-B2
26	N-R4	N-KN4
27	N-B5	R-QB2
28	P-N4	K-KN3
29	R-Q6	B-K7
30	R-Q8	BXP
31	R-KN8ch	K-KR4
32	N-N7ch	K-KR3
33	N-B5ch	K-KR4

etc. and drawn by repetition

First Game Won by Computer in Tournament Competition, Game 3 Tournament 2, Massachusetts State Championship 1967

White Mac Hack VI Black—1510

1	P-K4	P-QB4
2	P-Q4	PXP
3	QXP	N-QB3
4	Q-Q3	N-B3
5	N-QB3	P-KN3
6	N-KB3	P-Q3
7	B-KB4	P-K4
8	B-KN3	P-QR3
9	O-O-O	P-QN4
10	P-QR4	B-R3ch
11	K-QN1	P-N5
12	QXP/Q6	B-Q2
13	B-KR4	B-N2
14	N-Q5	NXKP
15	N-QB7ch	QXN
16	QXQ	N-B4
17	Q-Q6	B-KB1
18	Q-Q5	R-B1
19	NXKP	B-K3
20	QXN!	RXQ
21	R-Q8 MATE	

A More Recent Game With Times For Computer Moves, Game 2, Tournament 3 Massachusetts Spring Amateur

White Mac Hac VI Computer Time in sec

Black Unrated

1	P-K4	BOOK	P-K4
2	N-KB3	BOOK	N-QB3
3	B-QN5	BOOK	P-QR3
4	BXN	BOOK	QPXB
5	O-O	BOOK	B-Q3
6	P-Q4	BOOK	B-KN5
7	PXP	BOOK	BXN
8	QXB	BOOK	BXP
9	P-QB3	BOOK	Q-R5
10	P-KN3	18.3	Q-K2
11	R-K1	44.9	P-KR 4
12	P-KR4	111.7	O-O-O
13	B-KN5	78.5	P-B3
14	B-KB4	74.5	P-KN4
15	BXB	45.3	QXB
16	PXP	41.5	PXP
17	Q-KB5ch	60.5	QXQ
18	PXQ	29.1	N-B3
19	P-QB4	33.8	P-R5
20	N-QB3	77.6	R-Q7
21	P-QN3	88.0	P-R6
22	N-K4	56.0	NXN
23	RXN	43.3	K-Q2
24	P-KB6	19.0	R-Q3
25	P-KB7	19.0	R-B3
26	R-Q1ch	25.8	R-Q3
27	RXRch	22.8	PXR
28	K-KR2	68.25	R-KB1
29	KXP	76.6	RXP
30	R-K2	22.6	P-N4
31	K-N4	30.5	R-N2
32	R-K4	28.3	P-Q4
33	PXQP	19.4	PXP
34	R-K5	23.2	K-Q3
35	RXNP	14.0	RXR
36	KXR	4.5	K-K4
37	P-KB4ch	6.0	K-K5
38	P-KB5	8.5	P-Q5
39	P-B6	4.9	P-Q6
40	P-KB7	5.1	P-Q7
41	P-B8=Q	12.1	P-Q8=Q
42	Q-QB5ch	27.7	K-K6
43	Q-K6	29.2	K-B7
44	QXRP	42.8	Q-Q4ch
45	K-B4	20.45	Q-Q5ch
46	K-B5	14.9	Q-Q4ch
47	K-KN4	21.6	Q-KB6ch
48	K-KR4	16.3	QXKNPch
49	K-KR5	4.7	Q-K4
50	K-R6	20.8	Q-R1

etc. finally drawn by repetition

ACKNOWLEDGMENT

Many thanks go to the people at Project MAC who have written various routines, assisted in debugging the program by playing it, and served as operators at tournaments.

REFERENCES

1 C E SHANNON
Programming a digital computer for playing chess
Philosphy Magazine vol 41 March 1950 pp 356-375

2 A M TURING
Faster than thought B V Bowder (Ed)
Putman, London 1953 pp 288-295

3 A NEWELL J C SHAW H SIMON
Chess playing programs and the problem of complexity
IBM Journal of Research and Development vol 2
October 1958 pp 320-335

4 G W BAYLOR H A SIMON
A chess mating combinations program
Proc. Spring Joint Computer Conference vol 28 April 1966 pp 431-447

5 A KOTOK
A chess playing program for the IBM 7090
Bachelor's Thesis, Department of Electrical Engineering MIT 1962

6 G M ADELSON-VELSKY V L ARLASAROV A G USKOV
Programme playing chess
Report on Symposium on Theory and Computing Methods in the Upper Mantle Problem

7 P SAMSON
MIDAS
Artificial Intelligence Project Memo 90 MIT October 1965

8 A BERNSTEIN M DE V ROBERTS T ARBUEKLE M A BELSKY
A chess playing program for the IBM-704 computer
Proc. 1958 Western Joint Computer Conference Los Angeles Calif pp 157-159

9 J KISTER P STEIN S ULAM W WALDEN M WELLS
Experiments in chess
Journal of the ACM vol 4 April 1957 pp 174-177

1967 FALL JOINT COMPUTER CONFERENCE COMMITTEE

Chairman

L. C. Hobbs, Hobbs Associates, Inc.

Administrative Vice Chairman

David F. Weinberg
TRW Systems

Technical Vice Chairman

Sei Shohara
Scientific Data Systems

Technical Program

Chairman:
Harry T. Larson
Philco-Ford Corporation
Vice Chairman:
Ralph B. Conn
Aerospace Corporation
Jerry L. Koory
Programmatics, Inc.
Eldred C. Nelson,
TRW Systems
Frank C. Rieman,
TRW Systems
J. Seidman,
TRW Systems
Richard B. Talmadge
IBM Corporation
Edward O. Boutwell, Jr.
Compata, Inc.
Bruce Kaufman
Consultant
Robert E. Perry
Hughes Aircraft Company

Secretary

Irene E. Matthews
TRW Systems

Treasurer

Ronald Higgins
Teledyne, Inc.*

Exhibits

Chairman:
Samual F. Needham
TRW Systems
P. P. Gehl
The Marquardt Corporation
B. Green
TRW Systems
P. R. Lipinski
TRW Systems

Public Relations

Chairman:

James L. Pyle
California Computer Products, Inc.**
Gary Wormser
Electro Optical Systems
Dick Roper
Computer Sciences Corporation
Irwin Schorr
System Development Corporation
Charles Elkind
WEMA
Larry Bishop
Beckman Instruments
Joseph A. Sandy
Burroughs Corporation

Local Arrangements

Chairman:
H. H. Sarkissian
S. S. and S. Co.

Vice Chairman:
Richard D. Blosser
Autonetics
Dan Adams
TRW Systems
W. S. Dorsey
Union Oil Company
Lynn H. Maxson
IBM Corporation

*Replaced Richard P. Blunk, Teledyne, Inc., transferred out of Southern California

**Replaced Richard A. Russack, IBM Corporation, transferred out of Southern California

James E. Fallon
Capitol Records
Howard Verne
Hughes Aircraft Company
Frank F. Jurkovich
Informatics, Inc.
A. E. Olson
Douglas Aircraft Company
Pat Riley
TRW Systems

Printing and Mailing

Chairman:
Robert L. Koppel
Autonetics
Vice Chairman
Glenn W. Murray
Autonetics

Educational Program

Chairman
Gloria M. Silvern
Computer-Assisted Instruction Systems

Vice Chairman
Leonard C. Silvern
Education and Training Consultants
M. Jack Rand
Temple City Unified School District
Carl F. Heinz
Compton Union High School District
Jerry C. Garlock
Educational Testing Service

Science Theater

Chairman:
Charles H. Richards
System Development Corporation
Richard J. Werner
System Development Corporation
William F. Gallagher
System Development Corporation

Community Relations

Chairman
Robert B. Forest
Datamation
Richard H. Hill
Informatics, Inc.
Irving R. Bengelsdorf
Los Angeles Times
Janet Eiler
Datamation

Registration

Chairman:
S. B. Yochelson
Hughes Aircraft Co.

Vice Chairman
J. D. Anderson
Hughes Aircraft Co.
R. Breece
North American Aviation
T. Bohen
Autonetics
D. Theis
Hobbs Associates, Inc.
A. A. Perez
Lockheed Aircraft Corporation
M. Batchelder
Lockheed Aircraft Corporation
M. Heaton
North American Aviation

Entertainment

Chairman:
Stan Friedman
Lockheed Electronics Company
Lila Arner
Lockheed Electronics Company
Harry Fisher
Ampex Corporation
John Fisher
Radio Corporation of America
Robert Miller
Lockheed Electronics Company
Art Wroobel
International Rectifier

Tours

Chairman:
Alex M. Bradley
Varian/Data Machines
Muriel Gustin
Varian/Data Machines
Earl Brockman
Douglas Aircraft Co.
Jerry Harrer
Douglas Aircraft Co.
Jan Berkey
City of Los Angeles
J. F. Gloyderman
North American Aviation
Clyde Cornwell
Lockheed Electronics Company
Diana Midlam
Scripps Institute of Oceanography

Women's Activities

Chairman:
Ann L. Rataichak
IBM Corporation

Vice Chairman:
Mrs. James O. White, Jr.
Lockheed Aircraft Corporation

Joanne Kloepfer
Astrodata, Inc.

Mrs. Fred W. Springe
Autonetics

AFIPS Adviser

Keith W. Uncapher
The RAND Corporation

ACM Representative

Robert W. Rector
Informatics, Inc.

ADI Representative

L. H. Linder
Philco-Ford Corporation

AMTCL Representative

George Motherwell
Autonetics

IEEE Representative

Richard Tanaka
California Computer Products, Inc.

SCI Representative

Allan N. Wilson
Convair

REVIEWERS

C. T. Abraham
M. E. Alberda
R. Alonso
E. B. Altman
P. Armer
G. N. Arnovick
J. D. Aron
R. Asendorf
M. M. Astrahan
P. Atherton
A. Avizienis
P. R. Bagley
R. Barron
A. Bartholomay
R. S. Barton
S. Bauer-Mengelberg
J. Bayleff
G. Beckwith
M. F. Berman
M. L. Berman
P. Berning
L. Bernstein
W. P. Bethke
D. V. Black
J. Bloomfield
H. Blowey
G. R. Bolton
H. Borko
A. Bradley
C. Brague
R. Brennan
E. R. Brooks
W. Brunner
D. Burbeck
H. Burks
C. Burns
L. L. Burns
B. Bussell
L. D. Cady
P. Calingaert
D. Callender
H. Campaigne
A Campi
D. G. Cantor
R. L. Carmichael
W. C. Carter
T. E. Cheatham, Jr.
B. Cheydleur
C. Chong
W. F. Chow
C. W. Clewlow
L. Clingman
A. B. Clymer
J. B. Cohen
T. Condon
R. N. Constant
U. C. Dilks
S. Drezner
J. J. Dulin
E. Dunston
T. J. Dylewski
H. P. Edmundson
R. F. Elfant
A. W. England
G. A. Fedde
R. Filep
C. Fluke
F. H. Fowler
W. D. Frazer
J. Friedman
R. H. Fuller
L. Gainen
W. A. Ganzel
H. L. Garner
R. K. Gerlach
T. J. Gilligan
A. Goodman
N. R. Goodman
J. A. Gosden
M. Gotterer
A. Gradwohl
J. N. Graham Jr.
B. Gussel
T. Hagan
M. Halpern
A. G. Hanlon
P. Hart
H. P. Hartkemeier
E. Hartsfield
D. Hartwick
J. Harvey
C. H. Haspel
R. M. Hayes
R. Henle
P. Hickey
L. Hiller
L. Hirsch
R. W. House
R. M. Howe
A. Hughes
T. Hunter
C. C. Hurd
G. P. Hyatt
E. L. Jacks
R. E. Jackson
T. G. Jones
L. Justice
T. Kallner
W. Karplus
R. V. Katter
G. W. Kimble
K. King
L. Kleinrock
R. C. Knepper
M. Kochen
H. R. Koen
F. T. Krogh
R. L. Kuehn
D. J. Lasser
P. Lazarus
D. C. Lincicome
C. R. Lindholm
A. Ling
R. T. Loewe
H. A. Long
J. T. Lundy
R. E. Lynch
W. A. Marggraf
M. Maron
T. Martin
C. H. Mays
J. McCarthy
M. E. McCoy
J. McNeley
M. J. Merritt
S. W. Miller
R. G. Mills
Y. Mintz
E. Mitchell
G. Mitchell
O. Mock
M. Montalbano
G. Moshos
J. J. Murphy

J. A. Narvel
D. Nee
I. D. Nenama
R. Nelson
A. Newell
F. Newman
J. Noe
J. A. O'Brien
E. E. Osborne
J. J. Pariser
J. R. Parsons
L. Peterson
S. R. Petrick
N. A. Phillips
W. J. Plath
N. Pobanz
A. Pohm
D. A. Pope
J. A. Postley
A. W. Potts
J. P. Pritchard
J. E. Randall
K. Rash
L. C. Ray
S. Reed
A. M. Rees
R. Rice
P. A. Richmond
J. Rigney
L. G. Roberts
J. J. Robinson
D. E. Robison
C. A. Rosen
A. Rosenberg
R. F. Rosin
J. D. Sable
L. Sashkin
T. R. Savage
F. R. Schmid
J. D. Schmidt
A. Schneider
J. B. Schwarz
C. Sealander
S. Y. Sedelow
C. I. Seeley
W. Semon
D. Shansky
J. C. Shaw
G. T. Shuster
R. Silver
L. C. Silvern
Q. W. Simkins
R. F. Simmons
J. Sklansky
P. C. Smith
R. V. Smith
W. Smith
R. Spinrad
W. Spring Jr.
T. Standish
L. Stark
W. A. Stein
C. E. Stewart
T. G. Stockham
J. C. Strauss
W. Sturm
A. Svoboda
J. Sweringer
R. I. Tanaka
A. Teplitz
W. P. Timlake
H. D. Toombs
A. Toth
R. E. Turnage
R. Van Horn
H. Van Voeren
E. G. Vesely
R. Vichnevetsky
S. Viglione
P. Vlahos
R. V. Wakerling
C. A. Walton
H. R. Warner
W. Washington
G. A. Watson
B. D. Waxman
M. Weindling
E. H. Wentzin
G. P. West
G. E. Whitham
R. Willey
D. A. Williams
A. W. Wilson
C. Wimberley
J. E. Wolle
H. Wolpe
S. Wong
J. H. Worthington
R. E. Wyllys
J. W. Young Jr.
N. Zimbel

SESSION CHAIRMEN

L. D. Amdahl
G. A. Bekey
M. I. Bernstein
T. H. Bonn
M. A. Breuer
L. D. Cady, Jr.
A. J. Critchlow
J. D. Erdwinn
G. Estrin
T. Finch
R. M. Franklin
E. M. Grabbe
J. K. Hawkins
W. W. Herrmann
R. R. Johnson
P. J. Kiviat
R. G. Lex, Jr.
P. E. Parisot
M. Phister, Jr.
J. Raben
R. Rice
T. C. Rowan
A. I. Rubin
S. Sherr
J. W. Smith
L. M. Spandorfer
T. B. Steel, Jr.
E. G. Vesely
I. L. Wieselman
R. C. Wood

PANELISTS

F. Afferton III
J. D. Babcock
C. K. Bedient
A. Blumstein
R. Bryan
E. V. Comber
B. Creech
G. J. Culler
R. M. Davis
G. H. Dobbs
P. H. Dorn
R. S. Dunn
J. E. Engelberger
S. R. Erdreich
D. Evans
D. E. Farina
G. A. Fedde
D. A. Forman
W. B. Fritz
E. H. Gibbons
K. W. Goff
M. J. Halstead
E. Hartsfield
R. Head
M. Howe
M. R. Irwin
B. Johnson
R. C. Jones
R. C. Lawlor
T. A. Longo
R. E. Lord
P. Low
G. W. Markham
R. M. McClure
E. C. McIrvine
M. Minsky
N. J. Nilsson
S. Nissim
R. H. Norman
J. J. Pariser
R. J. Petschauer
C. Poland
K. Poovey
J. Priest
W. Quirk
T. Reddin
S. Rothman
J. E. Sammet
W. Sander
Q. W. Simkins
R. C. Simon
W. E. Simonson
D. Skoler
J. C. Strauss
V. A. Van Praag
F. V. Wagner
T. J. Williams

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES (AFIPS)

OFFICERS and BOARD of DIRECTORS of AFIPS

President

DR. BRUCE GILCHRIST*
IBM Corporation
Data Processing Division
112 East Post Road
White Plains, New York 10601

Secretary

MR. MAUGHAN S. MASON
Dept. 210
IBM Corporation—FSD
P. O. Box 1250
Huntsville, Alabama 35805

Vice President

MR. PAUL ARMER*
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Treasurer

MR. WILLIAM D. ROWE*
Sylvania Electronics System
189 B. Street
Needham Heights, Massachusetts

ACM Directors

DR. ANTHONY G. OETTINGER
Computer Laboratory
Harvard University
Cambridge, Massachusetts 02138

DR. ROBERT W. RECTOR*
Informatics, Inc.
5430 Van Nuys Boulevard
Sherman Oaks, California 91401

MR. J. D. MADDEN
ACM Headquarters
211 East 43rd Street
New York, New York 10017

DR. WALTER HOFFMAN
Computing Center
Wayne State University
Detroit, Michigan 48202

IEEE Directors

MR. SAMUEL LEVINE
Bunker-Ramo Corporation
445 Fairfield Avenue
Stamford, Connecticut 06902

MR. KEITH W. UNCAPHER
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

DR. T. J. WILLIAMS
Control & Information Systems Lab.
Purdue University
Lafayette, Indiana 47907

DR. R. I. TANAKA
California Computer Products, Inc.
305 N. Muller Street
Anaheim, California 92803

Simulation Councils Director

MR. JOHN E. SHERMAN*
Lockheed Missiles & Space Corp.
D59-10; B-151
P. O. Box 504
Sunnyvale, California 94088

American Documentation Institute Director

MR. HAROLD BORKO
Systems Development Corp.
2500 Colorado Avenue
Santa Monica, California 90406

Association for Machine Translation and Computational Linguistics-Observer

DR. DAVID G. HAYS
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Special Libraries Association

MR. BURTON E. LAMKIN
Library & Information Retrieval Staff
Federal Aviation Society
800 Independence Avenue, S. E.
Washington, D. C. 20003

*Executive Committee

AFIPS Committee Chairmen

Abstracting

DR. DAVID G. HAYS
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Admissions

MR. WALTER L. ANDERSON
General Kinetics Inc.
2611 Shirlington Road
Arlington, Virginia 22206

Awards

DR. ARNOLD A. COHEN
UNIVAC
2276 Highcrest Drive
Roseville, Minnesota 55113

Conference

DR. MORTON M. ASTRAHAN
IBM Corporation—ASDD
P. O. Box 66
Los Gatos, California 95030

Constitution & By-Laws

MR. MAUGHAN S. MASON
Dept. 210
IBM Corporation—FSD
P. O. Box 1250
Huntsville, Alabama 35805

Education

DR. MELVIN A. SHADER
IBM Corporation—SDD
1000 Westchester Avenue
White Plains, New York 10604

Finance

MR. WALTER M. CARLSON
IBM Corporation
Old Orchard Road
Armonk, New York 10504

Processing Technology

MR. HERB KOLLER
EBS Management Consultants, Inc.
1625 Eye Street, N. W.
Washington, D. C. 20006

Technical Program

MR. JACK ROSEMAN
Heliodyne Corporation
1401 Wilson Boulevard
Arlington, Virginia 22209

Government Advisory

DR. HARRY HUSKEY
University of California
Division of Natural Sciences
Santa Cruz, California 95060

Harry Goode Memorial Award

MR. WILLIS H. WARE
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

IFIP Congress 68

DR. DONALD L. THOMSEN, JR.
IBM Corporation
Old Orchard Road
Armonk, New York 10504

International Relations

DR. EDWIN L. HARDER
1204 Milton Avenue
Pittsburgh, Pennsylvania 15218

Planning

DR. JACK MOSHMAN
EBS Management Consultants, Inc.
1625 Eye Street, N. W.
Washington,D. C. 20006

Public Relations

MR. ISAAC SELIGSOHN
IBM Corporation
Old Orchard Road
Armonk, New York 10504

Publications

MR. STANLEY ROGERS
P. O. Box R
Del Mar, California 92014

Social Implications of Information

Information Dissemination

MR. GERHARD L. HOLLANDER
Hollander Associates
P. O. Box 2276
Fullerton, California 92633

Consultant

MR. HARLAN E. ANDERSON
Time, Inc.
Time & Life Building
New York, New York 10020

Newsletter

MR. DONALD B. HOUGHTON, 15-W
Westinghouse Electric Corporation
3 Gateway Center, Box 2278
Pittsburgh, Pennsylvania 15230

U. S. Committee for IFIP ADP Group

MR. ROBERT C. CHEEK
Westinghouse Telecomputing Center
P. O. Box 8839
Pittsburgh, Pennsylvania 15221

JCC General Chairmen

1967 *FJCC*

MR. L. C. HOBBS
Hobbs Associates, Inc.
P .O. Box: 686
Corona del Mar, California 92625

1968 *FJCC*

DR. WILLIAM H. DAVIDOW
Dymec Division
Hewlett Packard Company
395 Page Mill Road
Palo Alto, California 94306

1968 *SJCC*

DR. A. S. HOAGLAND
IBM Research Center
P. O. Box 218
Yorktown Heights, New York 10598

AFIPS Executive Secretary

MR. H. G. ASMUS
AFIPS Headquarters
9th Floor
345 East 47th Street
New York, NewYork 10017

1967 FJCC LIST OF EXHIBITORS

Adage, Inc.
Addison-Wesley Publishing Company, Inc.
Addressograph Multigraph Corporation
Amp, Inc.
Ampex Corporation
Anderson Jacobson, Inc.
Anelex Corporation
Applied Data Research, Inc.
Applied Dynamics, Inc.
Applied Magnetics Corporation
Audio Devices, Inc.
Auerbach Corporation
Auto-Trol Corporation
Bell System
Benson-Lehner Corporation
Bolt Bernaek and Newman, Inc.—
Data Equipment Division
Bryant Computer Products
Burroughs Corporation
Business Supplies Corporation of America

California Computer Products, Inc.
Calma Company
Canoga Electronics Corporation
Cheshire—a Xerox company
Comcor, Inc.
Compat Corporation
Computer Communications, Inc.
Computer Design Publishing Corporation
Computer Sciences Corporation
Computer Test Corporation
Computerworld
Computron Inc.
Concord Control Inc.
Conrac Division, Conrac Corporation
Consolidated Electrodynamics Corporation
Control Data Corporation
Corning Glass Works
Cybetronics, Inc.

Data Communications Devices, Inc.
Data Disc Inc.
Datamation
Data Processing Magazine
Data Products Corporation
Di/An Controls, Inc.
Digital Development Corporation
Digital Devices, Inc.
Digital Equipment Corporation
Digital Logic Corporation
Digitronics Corporation

Eastman Kodax Company
E-H Research Laboratories, Inc.
Electronic Associates, Inc.
Electronic Memories

Fabri-Tex Inc.
Fairchild Semiconductor
Ferroxcube Corporation
Friden, Inc.

General Computers, Inc.
General Electric Information Systems Marketing
General Kinetics Inc.
Geo Space Corporation

The Gerber Scientific Instrument Company
Hewlett-Packard Company
Holt, Rinehart and Winston
Honeywell, Computer Control Division
Houston Omnigraphic Corporation

IBM Corporation
Informatics Inc.
Information Control Corporation
Information Displays, Inc.
Interdata
ITT Industrial Products Division

Kennedy Company
Kleinschmidt, Division of SCM Corporation

Lenkurt Electric Company, Inc.
Link Group, General Precision, Inc.
Litton Industries, Datalog Division
Lockheed Electronics Company

McGraw-Hill Book Company
3M Company
The MacMillan Company
Magne-Head, a Division of
General Instrument Corporation
Mauchly Associates
Memorex Corporation
Memory Technology Inc.

Microswitch, a Division of Honeywell
Midwestern Instruments/Telex
Milgo Electronic Corporation
Mohawk Data Sciences Corporation
Morrissey Associates, Inc.
The National Cash Register Company
North American Aviation, Inc./Autonetics Div.

Patwin Electronics
Potter Instrument Company, Inc.
Precision Instrument Company
Prentice-Hall, Inc.
Programmatics Inc.

Raytheon Company
Raytheon Computer
RCA Electronic Components & Devices
RCA EDP Division
Redcor Corporation
Remex Electronics
Rixon Electronics, Inc.
Rotron Manufacturing Company, Inc.

Sanders Associates, Inc.
Scientific Control Corporation
Scientific Data Systems
Simulators, Inc.
Software Resources Corporation
Soroban Engineering, Inc.
Spartan Books
Spatial Data Systems, Inc.
Standard Computer Corporation
Sylvania Electric Products Inc.
Systems Engineering Laboratories, Inc.
Systron-Donner Corporation

Tally Corporation
Tasker Industries
Technical Measurement Corporation
Teletype Corporation
Texas Instruments Inc.
Thin Film Inc.
Thompson Book Company, Inc.
Trans-Controls, Inc.
Transistor Electronics Corporation
Tymshare, Inc.

UNIVAC
University Computing Company
Uptime Corporation
URS Corporation
U. S. Magnetic Tape Company
Varian Data Machines
Vermont Research Corporation
Western Telematic Inc.
The Western Union Company
John Wiley & Sons, Inc.
Wyle Laboratories Products Division
Xerox Corporation
Zeltex, Inc.

AUTHOR INDEX