Diss. ETH No. 7646

Lilith:
A Workstation Computer for Modula-2

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

(ETH Zürich)

for the degree of
Doctor of Technical Sciences

presented by
RICHARD STANLEY OHRAN
BSEE, BYU
MS, Univ of Utah
born June 11, 1942
citizen of the USA

accepted to the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. C. A. Zehnder, co-examiner

Zürich 1984

Diss. ETH No. 7646

Lilith: A Workstation Computer for Modula-2

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

(ETH Zürich)

for the degree of
Doctor of Technical Sciences

presented by
RICHARD STANLEY OHRAN
BSEE, BYU
MS, Univ of Utah
born June 11, 1942
citizen of the USA

accepted to the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. C. A. Zehnder, co-examiner

Zürich 1984

Druckort: D-DRUCK-SPESCHA, CH-9014 St.Gallen

Preface

This dissertation is based upon the author's participation in a research project conducted at the Institut für Informatik of the Swiss Federal Institute of Technology(ETH) in Zurich. The principal investigator of the project was Professor Niklaus Wirth. The project has been underway for 7 years now, and many people have contributed to it. The success of the project has exceeded most of our expectations.

Direct support for this project has come primarily from the ETH. My participation in the project has also been supported by the Department of Electrical Engineering of Brigham Young University which provided a sabbatical year of leave during which the project was initiated. The Department of Computer Science of Brigham Young University has also given support by way of a lightened course load and professional development leave. By virtue of the technology which it has distributed in behalf of the ETH and through grants from the Burroughs Corporation, the Modula Research Institute of Provo, Utah has also been able to provide financial support for my efforts in this project.

To all of my collegues on the project, I owe a debt of gratitude for the many enjoyable associations I experienced during the project and for the many enlightening converstations.

For many enjoyable hours working together on the construction of the hardware of the Lilith, I also thank Jirka Hoppe, Immo Noack, Farrell Ostler, Lyle Bingham, and John Nielsen.

To my friends and collegues, Duane B. Call and Aurel Soceneantu, I express thanks for their continual encouragement during the years we have worked together.

To Prof. C. A. Zehnder I owe a special debt of gratitude for his efforts on my behalf in solving the special bureaucratic problems associated with finding a position for a foreigner at the ETH, and for his help with this dissertation.

Two important lessons that I have learned from Prof. N. Wirth are the value of simplicity in technical matters and a proper perspective for the role of a professional career in life. For his profound influence on my professional career, I offer him my most heartfelt thanks.

The patience and sacrifices of all my family members have made it possible for me to finish this work.

Contents

Abstract

Lilith is a single-user workstation computer optimized for the development and execution of Modula-2 programs. Considerable departures from conventional techniques were made in its design to optimize the processor for the execution of this high-level language. Also, special features were included in the design to give the system extra versatility for its intended role as a programming workstation. These features include the attachment of a high resolution bitmapped graphics display and the inclusion of special operations in the processor which manipulate this display. A mouse pointing device was selected as a peripheral to the processor to provide good human interaction capabilities.

Their are three focal points of this dissertation, to document the design of the Lilith, to analyze the performance improvements which have resulted from various decisions made in the design, and to make recommendations for further improvements.

The selection of a stack based architecture is one factor contributing to the performance of the Lilith. It allowed the creation of a very compact instruction set (called M-codes) which has reduced considerably the size of Lilith programs in machine language, and thus contributed to the overall speed of execution. The inclusion of a fast stack for expression evaluation provides rapid execution of instructions. Dynamic measurements show that the majority of M-code instructions execute in fewer than five microcycles.

The organization of Lilith's memory is another factor contributing to the machine's performance. The use of a 64 bit read path allows the Lilith display controller to display a large number of bits on the screen (30 million bits per second) without appreciably interfering with the processor's access to memory. The wide memory path also allows the use of an instruction fetch unit which prefetches M-codes so that less time is spent by the processor fetching instructions. The reduction in time spent fetching instructions decreases Lilith execution times by a factor of one third in comparison with conventional machines that spend half of their time fetching instructions.

The special characteristics of Lilith make it a perfect candidate for further expansion by increasing memory size, decreasing cycle times, and extending the architecture to multiple processors.

Zusammenfassung

Lilith ist ein Arbeitsplatzrechner für einen einzelnen Benützer, optimal ausgelegt zur Entwicklung und Ausführung von Modula-2-Programmen. Durch beträchtliche Abweichungen von konventionellen Lösungen wurde der Prozessor auf die Ausführung von Programmen in dieser höheren Sprache ausgerichtet. Weitere Besonderheiten im Systementwurf betreffen dessen Eignung als Arbeitsplatz des Programmierers. Dazu gehören der hochauflösende Graphik-Bildschirm mit direkter Basierung auf dem Arbeitsspeicher (bit map) sowie spezielle Prozessor-Operationen für Graphikarbeiten. Eine "Maus" dient dabei als Eingabegerät zum Prozessor für eine ergonomisch gute Dialoggestaltung.

Die Zielsetzungen dieser Dissertation sind dreifach, nämlich die Dokumentation des Lilith-(Hardware-)Entwurfs, die Analyse der Leistungsverbesserungen als Folge der verschiedenen Entwurfsentscheide sowie Empfehlungen für weitere Verbesserungen.

Die Wahl einer Stapel-(stack)Architektur im Gegensatz zu einer Multiregister-Maschine ist eine wesentliche Grundlage für die Leistungsfähigkeit der Lilith. Sie erlaubt die Schaffung eines sehr kompakten Befehlssatzes (sog. M-code), welcher den Umfang der Lilith-Maschinenprogramme bedeutend reduziert und damit zur generelen Erhöhung der Ausführungsgeschwindigkeit beiträgt. Die Entwicklung eines schnellen 16-Stufen-Stack zur Auswertung von Ausdrücken erlaubt die rasche Verarbeitung des M-code. Dynamische Messungen zeigen, dass die meisten M-code-Befehle in weniger als fünf Mikrozyklen ausgeführt werden.

Der Speicheraufbau ist ein weiterer Grund, der die Leistung der Maschine erhöht. Der Einsatz eines 64 bit breiten Lesepfads erlaubt dem Lilith-Prozessor, eine sehr grosse Zahl von Bits (30 Mbits/s) laufend auf den Bildschirm zu übertragen, ohne die übrigen Speicheraktivitäten des Prozessors merkbar zu beeinträchtigen. Der breite Lesepfad erlaubt auch den Einsatz eines besonderen Befehlsvorprozessors (instruction fetch unit), welcher M-code-Befehle vorausliest, damit der eigentliche Prozessor weniger Zeit dafür benötigt. Die Zeitreduktion beim Lesen der Befehle vermindert die Lilith-Ausführungszeiten um einen Drittel, verglichen mit konventionellen Maschinen, welche die Hälfte ihrer Zeit mit dem Lesen von Befehlen verbringen.

Die besonderen Eigenschaften der Lilith-Maschine machen diese zu einem idealen Kandidaten für weitere Ergänzungen mit grösserem Speicher, schnelleren Zykluszeiten und weiteren Prozessoren.

1 Introduction

1.1 The Lilith Project

In the fall of 1977 a project was begun at the Institut für Informatik of the Swiss Federal Institute of Technology(ETH) in Zurich under the leadership of Prof. N. Wirth. The goal of this project was relatively ambitious: the creation of a complete new computing instrument--a workstation computer-- based on total rework of the existing technology. The scope of the project included the design of a new language, the design of a new operating system, and the design of a new computer architecture. The language was called Modula-2, the operating system was called MEDOS-2, and the computer itself was later christened "Lilith."

The name "Modula" is a contrivance from the word "modular language". "Module" is the name of a key programming structure in the new language. The name "MEDOS" was concocted by its author who stated on one occasion that it was taken from the German phrase, "Mein Erstes Disk Operating System", which means "my first disk operating system". The name "Lilith" came from Jewish mythology as the name of a female personage of questionable integrity who supposedly seduced men away from their wives and children.

There were many reasons for beginning such a project. One reason was a dissatisfaction with the architecture of computers previously used and the feeling that a new effort could yield a better result if a strong emphasis were placed on creating a design especially suited for high level programming languages. Another reason was a desire to experiment with new ideas for using computers in a mode where a relatively powerful computer would be dedicated to the use of a single programmer with the intention of providing better tools for the programmer to use. Finally, a number of years of experience in the use of high level programming languages had led to the discovery of new principles in the design of high level languages which could only be proven in the development and implementation of an entirely new language.

1.2 Technological Background

The design of the Lilith workstation was based primarily upon technology developed in several earlier projects at the ETH and upon work done at the Xerox Palo Alto Research Center. The concepts which were developed in these projects were the basis for creation of Lilith.

Pascal

The Pascal programming language [Amm] [Wir71] developed at the Institut für Informatik during the first part of the 1970's established the fundamental ideas of structured program languages which determined the architecture of Lilith. In particular, the techniques of storage addressing for variables, and the requirements for procedure calls in Pascal are directly reflected in the Lilith architecture.

P-Machine.

As acceptance of Pascal as a programming language became widespread, a mechanism was sought to allow an easier transfer of the programming language to other computer systems. A solution was found in the invention of a pseudo-machine architecture which could be simulated in the assembly language of the targeted computer system [NAJNJ]. The pseudo-machine was never referred to by any more elegant name than the P-machine. Its order codes were referred to as P-codes. It appears that its true significance as an innovative and extremely efficient computer architecture was never recognized. Even though the details of its design have been widely spread, in the intervening years not a single IC manufacturer has ventured to construct a single-chip microcomputer implementing the P-machine architecture. It has, however, been implemented on microprogrammed machines on several occasions [BS]. Lilith has drawn heavily from the ideas developed in the design of the P-machine.

Modula

As a precursor to Modula-2, the Modula programming language was developed "to conquer that stronghold of assembly coding, or at least to attack it vigorously" [Wir1] [Le]. In other words, the goal in its design was to completely eliminate the need for assembly language programming even at the systems programming level. The experience gained with Modula led to the inclusion of similar ideas in Modula-2 and the successful elimination of all programming at the assembly level in the Lilith computer.

The Xerox Alto

Exposure to the Alto computer system developed at the Xerox Palo Alto Research Center (PARC) [TMLSB] influenced the design of the human interface aspects of the Lilith system. The Alto made effective use of a high resolution *bitmapped* graphics display and of a pointing device called a *mouse*. These features were also incorporated into Lilith together with the technique of displaying characters and cursors created as raster images by software rather than hardware. The use of multiple font styles as an additional dimension for the presentation of information on the screen, and the use of overlapping windows as a further means of organizing information on the screen were techniques developed at PARC and carried over into the Lilith project.

1.3 Project Goals

Goals were formulated at the start of the project. These goals reflected, of course, the experience of the previous decade in the projects mentioned. For the most part these goals addressed themselves to shortcomings observed in the previous efforts and to some extent to aspirations encouraged by recent advances in semiconductor technology. At the time, no written document was generated enumerating these goals, so the following is a recollection of them as they came from the initial planning discussions.

The following are the goals discussed by the project team:

Rework and Extension of Pascal into a New Language: A language suitable for systems programming use as well as application programs was sought. Seven years of Pascal usage had unearthed many problems in the language. With the experiment of Modula completed (an experiment in the design of a language only for use as a systems programming tool), it was felt that the time was right to attack the general problem of a new language to replace Pascal for applications programming with augmented features to support systems programming.

High Level Language Programming: Having the conviction that assembly language programming was undesirable, it was hoped that the combination of a new powerful language and a correctly designed processor architecture could completely eliminate the need for assembly language programming. Even the operating system would be programmed in Modula-2 [Knu].

Stand-Alone Processor: It was hoped that sufficient processor power and memory storage capacity could be combined to provide a computing work station with adequate capability to support all language, operating systems and utility developments without the support of any larger computer system.

Efficient Programmer's Work Station Design: It was recognized that the use of a powerful processor coupled with a bitmap graphics display and a mouse pointing device would allow the development of more efficient methods of human interaction with the computer. This would be achieved through the versatility in the workstation's display and the speed with which images could be changed. By using the mouse for control operations, the programmer would be able to quickly issue commands to the computer and thus be capable of greater programming productivity.

Hardware Efficiency and Simplicity: Above all, we hoped to be able to achieve our goals with simpler hardware (relatively speaking). This was to be done by drawing upon the group's understanding of high level language compilation techniques so that a higher degree of efficiency in the processor would be achieved through the elimination of processor features unusable by high level language compilers.

Reliability and Maintainability: Naturally, we wanted the machine to be reliable and maintainable. We knew, however, that this goal would compete with previous goals of keeping the machine simple and efficient. For example, the addition of parity detection circuitry or error correction circuitry would have added complexity to the memory circuitry, but would have improved the reliability of the machine. Adding circuitry to allow automated sampling of the computer's internal signals would improve maintainability. If every opportunity were taken to add circuitry for the purpose of improving reliability or maintainability, the number of chips necessary to build the machine could easily be doubled. We hoped to find an optimal balance.

The Ideal Programmer's Workstation

In summary, it might have been said that our aspiration was to build nothing less than the ideal programmer's workstation. Indeed, each member of the group envisioned from the start the day when he would have his own copy of this machine in his office as the nearly perfect solution to his programming needs (or, as perfect as would reasonably be within our power to make it). Since all members of the group were extensively involved in the software side of computing, it was impossible to conceive that the machine would have any other orientation. Thus, Lilith was to become a programmer's workstation, optimized for all tasks relevant to programming, i.e. editing, document preparation, compilations, debugging, filing, etc.

1.4 Historical Notes

The following timetable outlines significant events in the Lilith project:

Oct 1977	Start of Project
Sep 1978	Completion of first (wirewrap) prototype at ETH--without disk
Sep 1979	Completion of second prototype (printed circuits)
Jan 1980	Start of 30 unit production at Brigham Young University in Provo, Utah
Jun 1980	Medos-2 operational on Lilith Compiler transferred from PDP-11
Oct 1980	Delivery of 10 units to ETH
Mar 1981	Delivery of 10 additional Liliths to ETH Announcement of Lilith [Wir81]
Sep 1981	Construction of 60 additional Liliths by Modula Research Institute in Provo, Utah
Jun 1982	Completion of the laser printer interface
Sep 1982	Completion of the network interface [Hop83]

Since the project's beginning in the last quarter of 1977, almost six years have passed. More than a hundred Liliths have been constructed and over 60 units have been installed at the ETH for use in the the Institut für Informatik (Dept. of Computer Science). Furthermore, Liliths have been supplied to numerous universities and corporations in North America and Europe for study of its software and hardware innovations. At least three different corporations have produced Liliths or variations thereof.

Initially the project was staffed by seven people: Prof. Niklaus Wirth, Project Director; Svend Knudsen, Christian Jacobi, Leo Geissmann, Jirka Hoppe, Werner Winiger and the author. Knudsen, Jacobi [Jac], and Geissmann worked on the compiler [Gei]. Knudsen programmed Medos-2 alone [Knu]. Winiger wrote the microprogrammed interpreter and worked on the editor utilities. Hoppe wrote most of the diagnostics [Hop2] and designed the display controller for the

15 inch landscape mode monitor. The author worked on the design and debug of the hardware. N. Wirth, in addition to supervising the project, also was the principal architect of the machine and its instruction set. He also programmed the text editor and the line drawing program, and designed interfaces to the laser printer and the portrait mode display.

In the summers of 1978 and 1979, the project construction and debug efforts were aided by three Brigham Young University students who came to Zurich for summer vacation: Farrell Ostler, John Nielsen, and Lyle Bingham.

In 1978, Immo Noack and Anthony Gourengourt joined the project. Gourengourt aided in the compiler effort while Noack aided the hardware side.

The areas of the Lilith project which were in part or wholely the contribution of the author were the design of the memory subsystem, the data port, the instruction fetch unit, the expression evaluation stack, the short constant masking circuitry, the system timing circuitry, the diagnostic processor, the disk interface, the priority interrupt circuitry, the backplane, and the packaging of the system.

1.5 A Scenario of Lilith Usage

The mere enumeration of a machine's statistical characteristics do not convey adequately the power and effectiveness one experiences when working with Lilith. Therefore, a hypothetical scenario of machine usage will be described as it was envisioned by the designers and as borne out in actual use of Lilith.

A Lilith user approaches the system bringing with him his own disk cartridge containing 10 megabytes of disk storage. This includes his personal text and program files, a complete library of systems software, demonstration programs, standard and/or specially prepared font files, documentation files and graphic illustration files. After he inserts his disk cartridge and powers up the system, a boot program is read into the main memory establishing a core of resident modules which are so commonly used that their presence as a"default" is felt necessary.

After a few seconds the user has already instructed the command interpreter to initiate execution of the document editor, and he resumes a previously begun session of editing a document. He views this document on a high resolution screen seeing characters very much like those of a typeset document. Some characters are larger, some are italicized, and others are bold. Using the mouse, he points to a paragraph and inserts characters into the text by typing on the keyboard. Effortlessly selecting a group of words with the mouse (by coordinated use of the buttons on the mouse and by moving the mouse), he then causes a small rectangular menu to appear in the middle of the screen by pressing another button. He quickly selects an option from the menu by pointing to it and releasing the button. The option causes the previously selected group of words to be repainted on the display in a bold typestyle. As he continues to type and give control commands with the mouse, he constructs for himself an image on the screen which corresponds to the document which he will later cause to be printed in the laser printer. The laser printer output appears at first glance to be flawlessly typeset only on closer inspection one recognizes that it is not quite as good as typesetting.

Later he works upon the improvement of a program, moving rapidly between the phases of editing, compiling, and execution. While editing, the screen is populated by windows which separate groups of information. In one window is the source code of another's program containing a portion of code usable for his current effort. With single mouse commands he moves this code from that window into the window containing his program and adapts it to his program.

While debugging an abnormally terminated program, the screen is populated with windows containing extracted information from the interrupted execution of his program. In one window he sees the compiler listing for the program correctly positioned to the point of interruption. In another window, he views data associated with the program tabulated with its symbolic name, type, and most recent value when the program execution was suspended. In all, six different windows are displayed, each having its own function associated with the debug of the erroneous program. Some of the windows have been expanded in size so that they overlap others but provide a larger viewing area. With a simple mouse operation, the windows are brought to the surface like sheets of paper scattered on a desk. The error of the program is easily found because the coordinated design of hardware and software in Lilith has given the programmer an exceptionally good capability to discern the causes of program malfunction.

As the programmer completes his work session, his final act is to place a backup copy of his work on the central file storage system which is accessible through the network linking many Liliths with other service units such as an image scanner, and a laser-printer. His last act is to remove his cartridge from the machine and to return it to his place of safety, confident that his own work effort will not be disrupted by central disk crashes, unexpected system software updates, or malicious users.

2 Overview of the System

The basic Lilith system consists of a processor, memory, high resolution bitmapped graphics display, 10-megabyte cartridge disk, and miscellaneous I/O devices. These include: mouse pointing device, keyboard, real time clock, and serial line receiver/transmitter interface (UART). A block diagram of the system is given in Figure 2.1.

2.1 Processor

The processor is a microprogrammed implementation of a stack machine architecture. It processes words of data which are 16 bits long. The most elementary processing step is a microinstruction which requires a period of 150 nanoseconds to execute. During the execution of a microinstruction, one or two words of data can be taken from the internal registers of the processor and operated upon in a monadic or dyadic fashion with the result subsequently being stored in another register. The operations which the processor can perform during a microinstruction treat data as integers, cardinals (positive integers), reals, addresses and sets of bits. There are also a class of operations which manipulate half-words; these are typically used with characters.

Sequences of operations executed by the processor are specified by instructions which the processor fetches from memory. The instructions may be one or more bytes long. These instructions are called M-codes. A single M-code may require the execution of many microinstructions in order to complete its execution. The Lilith processor typically executes about 1.2 million M-code instructions per second.

Some of the more distinctive features of the Lilith processor are the special operations which provide support for the use of the high resolution bitmapped graphics display. These operations are implemented in microcode and have especially efficient algorithms for the movement of images on the screen, for the creation of images on the screen, and for the display of characters on the screen. To help these graphics operations execute as rapidly as possibly, a special barrel shifter was designed into the arithmetic part of the processor.

The heart of the processor is a 16-bit arithmetic unit based on AMD2901 "bit slice" integrated circuits. The 2901 based arithmetic unit has 16 internal registers and has also been augmented with the barrel shifter mentioned in the previous paragraph and an external 16 level stack. Information flows into and out of the processor through the system bus which is 16 bits wide and connected to most subsystems of Lilith.

2.2 Memory

The memory of Lilith is constructed from conventional semiconductor memory chips of the dynamic refresh type. The cycle time is 375 nanoseconds in the fastest version of the machine. The most common configuration has 256k bytes (k = 1024), but versions of the machine have been built with larger memories.

The display and the processor compete for cycles from memory. The allocation of memory cycles is handled by a priority arbitration circuit. The display has the highest priority. For reading, the memory is organized basically as 32,768 addresses of 64-bit memory words. For writing it is organized into 131,072 sixteen-bit words. The 64 bit wide data path is not usable by all elements of the system. So, a special multiplexing circuit allows the memory to be read in 16-bit words. The arithmetic unit of the processor and the programmed I/O devices read memory in 16 bit words, but the instruction fetch unit of the processor and the display exploit the full 64-bit word length for their memory cycles.

For dynamic memory refresh, a memory port is allocated to a section of circuitry which periodically requests "dummy" memory read cycles. The circuitry requests these cycles to satisfy the requirement of the dynamic memory to refresh each of the 128 row addresses once each two milliseconds.

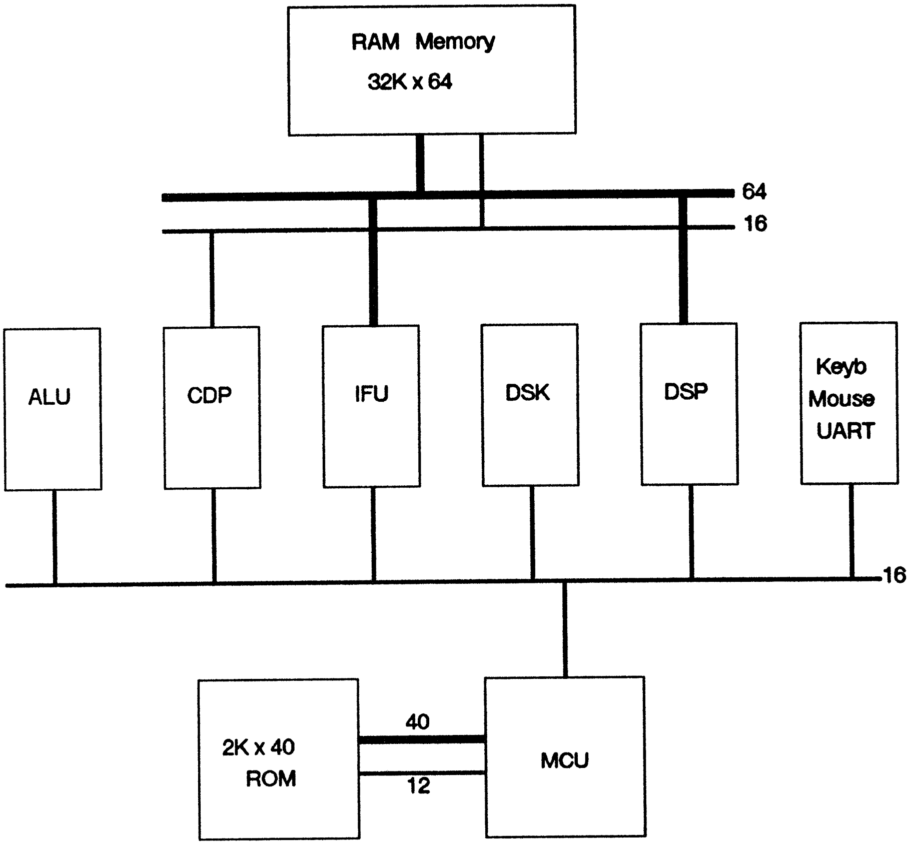


Figure 2.1 Lilith Block Diagram

2.3 Display

The display controller refreshes a high resolution 17 inch cathode ray tube from the main memory of Lilith. The display image is a so-called *portrait mode* display. The refresh rate is 36 frames per second, and the structure of the image is 704 dots across and 928 dots vertically. The total memory needed to store all dots of the screne and used to refresh the image includes 40,832 sixteen-bit words. All images displayed on the screen are constructed from "bitmap" images written into the display area of memory. The processor has several microprogrammed instructions which perform image-creation operations with greater efficiency than normal programming.

2.4 Disk

The secondary storage device of Lilith is a ten megabyte Honeywell Bull cartridge disk drive. The unit uses portions of the technology commonly referred to as "Winchester" yet provides flexibility in the form of a removable disk cartridge, 12" by 12" by 1". The disk controller has a minimal amount of logic, it is constructed on a single 8.5" by 11" circuit board, and it transfers a single sector of data to an onboard buffer of 128 sixteen-bit words. Data transfers to the main memory are not handled by direct memory access, but are handled directly by the processor using special microcoded routines which operate at full memory speed.

2.5 Mouse Pointing Device and Keyboard

One of the significant features of Lilith is its well-designed human interface facilitated by the previously described high resolution display, its "mouse" pointing device, and its keyboard. The mouse rests on the table and transmits relative surface movements to the processor. Software can use this information to position a pointer on the screen. Through pointing and depressing the three buttons on the mouse, the user can give commands to the processor which are typically program control operations. The effectiveness of this interface can hardly be described; it must be experienced.

2.6 RS-232c Interface

For communication with other standard peripheral devices and computers, a RS-232-compatible serial interface is provided. The transmission rate is selectable from 75 to 9600 bits per second. The voltage levels for mark and space are standard.

2.7 Real Time Clock

At the line frequency (50 Hz for Europe, 60 Hz for the USA) interrupts are generated which cause the processor to activate a "sleeping" task which keeps track of time and scans for keyboard entries.

2.8 Extra I/O Slots

For expansion of the machine's memory and connection of peripheral devices, the machine has additional uncommitted circuit board slots. The devices which are connected may be designed for programmed I/O or for direct memory access techniques using one of the four uncommitted memory ports.

3 The M-code Machine

Lilith was intentionally designed so that most people (i.e. programmers) who come in contact with it would never see anything but the Modula-2 as a means of programming the processor. In fact, many of the machine-dependent characteristics of Lilith which one could see at the Modula-2 level are typically not seen because they are buried in library programs where solutions of programming problems are provided to programmers at an abstract level free from concern about their intricate, albeit mundane, details of implementation. This insulation from real interaction with the hardware is considered to be one of the tenor themes in the philosophy of Lilith's design. The values of not having to program problem solutions on Lilith at the machine language level are proudly proclaimed. This does not, however, mean that Lilith does not have a machine language. In fact, it has two machine languages: M-codes which are fetched and executed from main memory, and microinstructions which are fetched and executed from read-only control store.

M-codes are the topic of this chapter. They are the molecules from which programs are built. As a program, a sequence of M-codes represents a translation in Lilith-executable form of an original Modula-2 program. The Modula-2 compiler must create this sequence of M-codes so that the semantic meaning of the original program is accurately reflected by the execution of these M-codes. It is the task of the compiler writer to see that his compiler accurately translates the Modula-2 programming constructions into valid M-code machine instructions. To do this, he must understand the nature of M-codes and the structure of the M-code machine which executes them.

It is also the task of the hardware designer to understand M-codes for he must build the computer so that in executing M-codes, his hardware correctly performs the computing functions assigned to each M-code. Because subsequent chapters will address the design of the Lilith hardware, it is necessary in this chapter to present the characteristics of the M-code engine which determined why and how the hardware of Lilith was designed.

3.1 An Overview of the M-code Machine

Like most computers, Lilith performs its task by the stepwise execution of a sequence of instructions which it has stored in its memory. The operation is cyclic in nature: first an instruction is fetched from memory, and then it is executed. Then the next instruction is fetched, and the process repeats itself. The instructions, called M-codes, may require as many as 5 bytes of memory for their specification, but most only require a single byte. The selection of the next M-code to be taken for execution (after the one being executed) is by default the M-code occupying the next adjacent memory location unless the current M-code being executed specifies a different sequence.

There are 256 M-codes which may be distinguished from each other by the contents of the first byte of the M-code. Most of the M-codes do not require any additional bytes; some require as many as four. M-codes requiring additional bytes typically use them for addressing offsets or immediate data. All together, the M-codes comprise the *instruction set* of the machine. The goal, of course, is to have a sequence of M-codes stored in memory stepwise directing Lilith to perform a sequence of operations which will produce a desired solution to a problem. How to create a sequence of M-codes which satisfies one's needs is the subject of the discipline of *programming*. We will not delve into that subject here; we will only attempt to describe the M-code instruction set available to those who do.

Arithmetic and Logic Operations

As with most computers, Lilith has an arithmetic/logic unit (ALU) which can perform operations upon words of data. The operations themselves are fairly conventional, i.e. 2's complement integer arithmetic, unsigned integer arithmetic, *real* arithmetic using floating point notation, and *boolean* operations of AND, OR, and XOR on sets of bits. Lilith differs from conventional machines in its use of a stack as both the source and destination for the operands and the results which are used and generated by the ALU. Conventional machines move data from memory to registers and their instruction sets specify which registers are to deliver their contents for use in the arithmetic/logic operations. In Lilith, data is moved from memory to the stack, and the top levels of the stack are always implicitly selected as the source for the operands of any arithmetic/logic operation. Furthermore, the result of any operation is always returned to the stack by implicit addressing. The use of a stack in this fashion is particularly efficient as has been described on numerous occasions in the literature [Bar] [Ham].

The stack used for arithmetic operations in Lilith can accept up to 16 words of data before any loss of data by *overflow* occurs. Other processors in the past which have incorporated stacks for arithmetic instruction evaluation have provided mechanisms for detecting when data has been lost by overflow. Some of them, such as the Burroughs computers and the Hewlett Packard computers, have even had mechanisms to automatically transfer data from an arithmetic evaluation stack to memory [Bur] [HP]. Lilith has no such facilities. It does not even have the capability in its hardware to detect whether or not the stack has overflowed. The reason for this departure from the conventional wisdom lies in the commitment of Lilith's designers to the use of high level languages. It is entirely possible to organize the use of the stack so that a compiler preparing the M-code program will always know to what extent the stack is being used. Therefore, if the compiler is working properly, it will never allow stack usage to exceed capacity.

Data Transfers between the Arithmetic Evaluation Stack and Memory

A further example of how high level language usage has directed the architecture of Lilith is found in the addressing mechanisms which the M-code instruction set offers for transferring data between main memory and the stack. The most striking thing about Lilith is what it does not offer. A conventional computer almost always has an instruction to load register **r** with the contents of memory location **m**-where **m** is a completely specified address of memory, or an abbreviated address of some section of the memory which is implicitly selected by default. Lilith has no such instruction to offer. All addressing of memory (for data purposes) in Lilith is done by reference to a base address register. This reflects the needs of programming in Modula-2 and other languages. Data variables as they exist in Modula-2 programs are typically organized as part of local or global data areas. Or, they are allocated from an area of the main memory known as a heap and their address is assigned as the contents of a *pointer* variable which is in the local or global data areas. Consequently, memory addressing in Lilith is effected by reference to base address registers which point to the beginnings of the local and global data storage areas together with an offset value which is included in the instruction as a constant. For pointer variable addressing, the contents of the pointer memory location are loaded onto the stack and the top of the stack is used as a base address register.

(**Note:** Interestingly, another computer which had this same characteristic of only addressing by base address register was the first eight bit microcomputer made, the INTEL 8008[Int2]. The direct addressing capability was later added to the improved version, the Intel 8080. I remember distinctly the feeling of puzzlement about how one would go about using the device. The answer, of course, lies in using immediate data operations to first load a register with a desired address before using the register to address the memory location. The same technique is usable in Lilith for the few instances that direct addressing is required.)

There are many advantages to addressing memory by use of base registers. Foremost is the compactness of the instruction set which can result. Lilith makes extensive use of instructions which have only 4 bit offsets as part of the instruction. This allows the complete specification of *load* operation from a word in local or global memory to require only a single M-code instruction byte.

M-code Addressing

When a location of memory is addressed as an M-code, its contents are taken as the next instruction to control the operation of the processor. This operation is called the *instruction fetch* and its performance is assigned to the *instruction fetch unit* (IFU) of the processor. Unless otherwise directed, the instruction fetch unit retrieves a sequence of M-codes from adjacent memory locations whose monotonically increasing addresses are supplied by a register within the IFU called the *program counter.* From time to time, one of the M-codes stipulates an alteration in the sequence of M-code fetches by loading a new value into the program counter. This is an operation typical of computers. The most common such operation is the *jump* operation which uses as an operand an address specifying which memory location is to be the beginning of the new sequence of instructions executed. Here again, the high level language orientation of the Lilith design departs from conventional techniques. Lilith has no direct jump. It does have a relative jump, which is to say that an M-code can instruct the IFU to go forward or backward an amount relative to its current position. But, an M-code cannot instruct the IFU to go to a specific memory location directly.

Direct jumps in Lilith are handled by tables of addresses. These addresses identify the beginning points of procedures. Since all programs written in Modula-2 may be viewed as collections of procedures, the only time a direct jump is required is when a procedure call is made. Within a procedure itself, all jumps are relative. Consequently, the only way Lilith offers a jump to an absolute address is by a procedure call which finds the address in a table.

All conditional branches perform relative jumps based upon operation results left on the top of the stack. For example, a comparison of two integers generates a boolean value of *true* or *false* which is left on the stack after the two integers have been removed. A conditional jump operation will then load a new address into the program counter depending upon whether it finds a *true* or *false* on the stack.

Procedures and Modules

In Modula-2 procedures and modules are constructs in Modula-2 having names by which the programmer knows them. In the M-code machine the procedures and modules are known by numbers and their locations in memory are found as addresses in tables indexed by these numbers. Procedures are always found within modules. Each module has one or more procedures within it. Therefore, each module has a procedure table. When a program makes a call to a procedure, it must first find the location of the module from the module table. Then, from the module's procedure table, it finds the starting address of the procedure.

A call to a procedure is similar to the subroutine calls typical of most computers. A return address is saved for later use when the procedure is completed, and the instruction fetch unit is instructed to continue program execution at the called procedure. Most computers perform this much. In Lilith, more work is necessary to satisfy the needs of Modula-2.

Each time a procedure is called or completed in Modula-2, the *scope* of local and global variables which are accessible changes. In the context of the M-code machine, this means that the base addressing registers used by Lilith may need to point to new data areas. For procedure calls within the same module, it is only necessary to change the local base address registers, but for calls to procedures within external modules, both global and local base address registers must be updated. Furthermore, access to variables at intermediate levels in the Modula-2 program structures may require the creation of a chain of pointers by which their access can be facilitated. A *procedure activation* record is the mechanism used to accomplish this. Each time a procedure is called, a procedure activation record is produced which saves the return address, the temporarily displaced contents of base address registers, and the link in the chain of pointers which provides access to the variables at intermediate levels.

The nested structure of procedure calls and the possibility of recursively calling procedures in Modula-2 dictates the need to use a stack within main memory as the mechanism for storing the procedure activation records. As procedures are called, the stack grows; as they are completed the stack shrinks. When procedures are called recursively, the stack grows and may have many activation records for the same procedure. The M-code machine maintains the highest used stack address in a register called the *stack register.* It uses this register for controlling the use of the stack within main memory. Each procedure call or completion requires the updating of content of this register. Fortunately, for the compiler writer, most of these operations are taken care of by the M-code machine with little management necessary.

Dynamic Storage Allocation

The higher extremity of the unused memory space allocated to the stack is available for allocation as the *heap*. When the program wishes to allocate storage dynamically during execution and assign this storage to a pointer variable, it obtains this storage from the top end of the stack. Naturally, the stack has fewer memory locations available for procedure activation records when a portion of its space is taken away and assigned to the heap. A base address register in the processor, the heap register, is used to keep track of memory so allocated. As the stack and the heap are used, the addresses pointed to by the stack register and the heap register grow closer together. If these two pointers meet, then the M-machine will terminate its current computing activity with an error indication of stack overflow.

Coroutines

As the M-code machine operates, it is sometimes desireable to suspend activity on a given computing task and momentarily perform some work in an unrelated computing function. In order to satisfy this need, the M-code machine has been given an instruction which will deposit into a small memory space all information about the machine state relevant to the current computing task. At some point later in time the machine state can be reloaded from the memory space, and execution of the program can resume with no difference distinguishable to the program. This operation actually consists of two steps. The first step is the saving of the program execution state into the memory locations called the *process descriptor*. The second step is the loading of a program execution state from another process descriptor, which has already been suspended. (Note: of course, there is a technique for creating a valid process descriptor for a computing task that has not yet begun execution. How could one otherwise have a process descriptor to transfer to?)

Having the ability to switch between independent computing tasks forces us to formulate a new concept for a computing task just as being able to travel across a body of water separating two areas of land forces us to formulate a new concept for land, i.e a continent. Each separate computing activity is called a *coroutine*, and switch by the M-code machine from execution of one coroutine to another is called a *coroutine transfer*.

Coroutine transfers can be programmed explicitly by a single M-code instruction. They can also be initiated by the hardware in response to requests from other elements of the system. Such requests are known as *interrupts*. When the M-code machine responds to an interrupt request, it suspends the current coroutine into a process descriptor pointed to by a fixed location of memory, and it activates a coroutine to handle the interrupt from a process descriptor pointed to by another fixed location in memory. If an interrupt occurs without valid process descriptors at the fixed locations of memory, the M-code machine will ignore it. From a programming standpoint, this mechanism for handling interrupts is elegantly simple.

Coroutines transfers are also initiated whenever the M-code machine discovers a run-time execution error such as arithmetic overflow, or invalid pointer addressing. In such cases, the processor looks to a fixed memory location for a valid process descriptor. If it finds one, it will be activated with the expectation that the coroutine stored there will undertake some error recovery action or else abort the computing task which created the error. Naturally, the M-code machine cannot assure that the coroutine activated will actually perform this function; it is the role of the programmer using the system to determine what a run-time error handling coroutine does.

M-codes for Special Purposes

Lilith has a number of M-codes which perform special functions not necessarily associated with the needs of programming in Modula-2. M-codes exist for functions such as high speed bitmap graphic screen manipulations, input/output operations, and preparation of images to be printed on a laser printer. These M-codes contribute greatly to the impressive performance Lilith gives as a programmer's workstation computer.

3.2 The Structure of the M-code Machine

The key elements of the M-code machine are the following:

BUS is a sixteen-bit data path which connects all elements of the machine together.

The *main memory* is a standard random access memory operating in read and write cycles with addressability to 256k sixteen-bit words.

The *global register* (G) and *local register* (L) provide the base addresses for address calculations needed for access to program data variables from main memory.

The *stack top register* (S) and the *heap register* (H) provide the address values used for managing sections of main memory as a stack.

The *code frame register* (F) and the *program counter offset register* (PC) compositely produce by addition the *program counter* register. Together they are used to generate a byte address for each M-code machine instruction. These registers are part of the *instruction fetch unit.*

The *process register* (P) holds a pointer to the process descriptor area for the currently executing coroutine.

The *mask register* (M) holds the interrupt mask which determines which interrupts may be recognized.

The *evaluation stack* is a true last-in-first-out stack structure composed of sixteen levels of sixteen-bit words. It is the source of operands and the destination of results for most operations carried out by the arithmetic/logic unit.

The *arithmetic/logic unit* operates on data values stored in the *evaluation stack.* It performs the arithmetic operations on the following data types: *cardinal*, *integer*,, and *real*. It also performs relational operations comparing magnitudes for these types yielding results of type *boolean*. It can perform the standard logic operations on *boolean* and *set* data types. For address calculations the *arithmetic/logic unit* may reference the global and local variable base address register as well as *BUS* for access to constants and offsets.

There are additional elements of the M-code machine which need to be introduced at this time. These elements are organizational in nature, being more a matter of usage convention within memory than of structure.

The *module frame table* is a table of pointers found in the memory of Lilith. Each program module has a base address entered for it into the *module frame table* so that it may be addressed by other modules.

The *data frame* is an area of memory assigned to a module holding its global variables and string constants. The address of the *data frame* is the value entered into the *module frame table.*

The *code frame* is an area of memory assigned to a module for its compiled instructions. The address of the *code frame* is the first word of the *data frame.*

The *procedure address table* is a table of values giving the entry point address for each procedure of a module. The entry point address is an *offset* amount from the start of the *code frame.*

A *process descriptor* is a data structure in the memory which can hold the state of a coroutine during the times when it is suspended.

The *interrupt vectors* are pointers to process descriptors which are assigned fixed memory locations. When an interrupt occurs, Lilith can create a coroutine transfer by going to the fixed addresses to find process descriptors where it can save the state of the coroutine in execution and retrieve the state of the coroutine which needs to be activated to handle the interrupt.

The overall relationship of the elements discussed in this section is shown in Figure 3.1. The relationship of these elements will be discussed in greater detail in following sections.

3.3 The M-code Machine Operation

As was briefly discussed previously, a computing task performed by the M-code machine is called a *coroutine,* and more than one coroutine can be operational in the machine at any given time although only one coroutine is actually in execution (called *active*) at any give time, while the others are held in a *suspended* state. Changes which cause the active coroutine to be suspended and replaced by another suspended coroutine are called *coroutine transfers. Coroutine transfers* may be caused either by programmed instructions or by hardware interrupts.

Each coroutine has a number of elements which make it unique and distinguish it from other coroutines active within the system: First, each coroutine is assigned a workspace in main memory, which it uses in part as a stack and in part as a heap. When the coroutine is created, the registers associated with the stack and heap are initialized appropriately to use the workspace. Figure 3.1 illustrates the address relationship of the S, L, and H registers to the workspace.

A second unique element of a coroutine is the *process descriptor*, which is a storage area capable of holding the entire state of the processor relevant to the execution of the coroutine. When a coroutine is suspended, its state is saved into its *process descriptor.* Usually, the storage for the process descriptor is placed at the beginning of the workspace by the procedure which initializes the coroutine and sets it into operation. A pointer to this area of memory is always held in the *P* register of the M-code machine for the active coroutine.

A third unique element of each coroutine is the program counter value which determines where--from program modules loaded in memory--the coroutine is to take its next instruction. It will be discussed later how this value is actually a combination of values from two of the machine's registers, F and PC. The program modules themselves are not uniquely assigned to any coroutine. They are simply resident in the machine and available to perform tasks for any coroutine which chooses to execute their instructions.

A fourth element unique to each coroutine is the local storage allocated for each procedure as the procedure is entered. This storage belongs to the coroutine which was active when the procedure was entered and is available for the use of that procedure only when **that coroutine** is active. If two coroutines enter and execute instructions from the same procedure, each will have its own separate copy of the local variable storage. This will not be the case for the global storage defined for each module and accessed by the procedures of the module. When such global variables are used by two coroutines executing from the same procedure, the storage referenced will be the same for both. This subject will be discussed further in the section describing module organization.

The following analogy may help the reader to clarify the conceptual relationship between coroutines and program modules as they exist in Lilith: We may view each coroutine as a moving, dynamic entity to be compared with an automobile. A program module in this analogy would be comparable to a neighborhood of streets, buildings and passages. The collection of all program modules in Lilith's memory is analogous to an entire city. Just as cars move through the city constrained by the configuration of the streets and pass from one neighborhood to another, coroutines work their way through the structures of program modules performing functions and moving on to other modules. Just as several cars may travel in similar paths or even travel down the same streets, several coroutines may operate within the same program module sharing global data but each privately using its own local data.

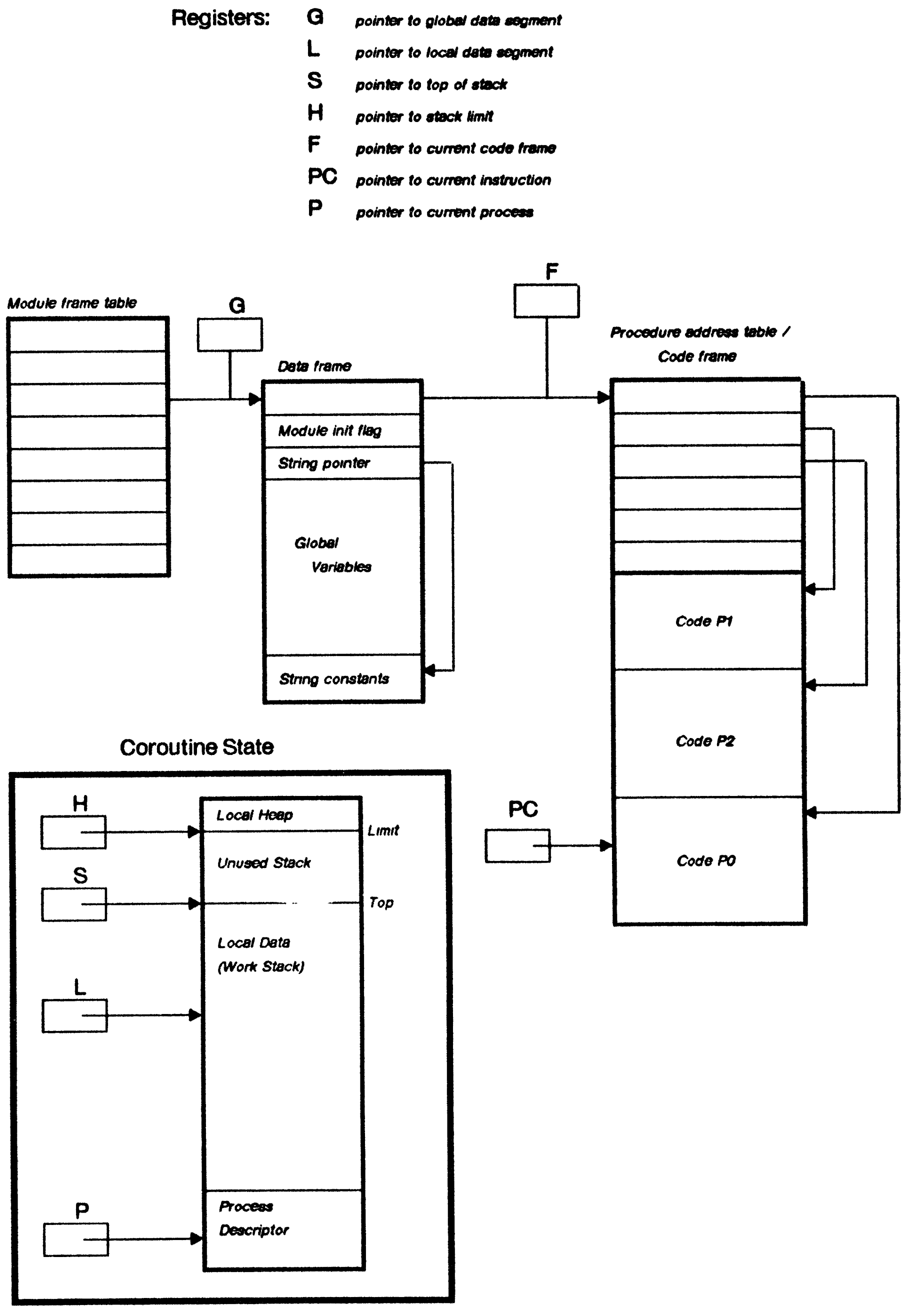


Fig 3.1 Module Organization in Memory

Instruction Fetch and Execution

Once the M-code machine has a valid coroutine in an active state (there is always at least the initial coroutine set up by the microprogram when the "reset" button is pushed), the M-code machine follows a repetitive pattern of fetching instructions and executing them. The basic cycle begins by fetching an M-code instruction from main memory. The address of an M-code is a so-called *byte address* because an instruction may consist of as little as 8 bits taken from either the right or left half of a normal 16-bit word of memory. Hence, a *byte address* requires one additional bit in the least significant position of the address to determine from which half of the word the address is taken. This *byte address* is computed from the sum of the contents of the code frame register (F register) and the program counter offset register (PC register). The F register value is shifted left two bits and the PC register value added to it; therefore, the resulting address has 18 bits. An 18-bit byte address allows addressing of 256k bytes of memory or 128k words (k = 1024) . The so-computed *byte address* is passed to the memory and the selected instruction byte is returned from the memory to the M-code machine.

After the instruction byte has been fetched, the program counter offset register is incremented by one. If additional bytes are needed to complete the instruction, they will also be fetched in the above fashion with the possible exception that if the decoding of the first byte specifies that the second byte is to be fetched as a negative value (for example, backward jumps), then the byte is return as a negative number.

Once the instruction byte has been decoded, and additional instruction bytes have been fetched as needed, the processor enters the execution phase wherein the specified operation is carried out according to the order code of the instruction.

The M-code Instruction Set

Eight bits, i.e. one byte, constitute the basic element from which instructions are composed. A complete Lilith instruction may use one through five bytes. (Refer to Figure 3.3.) The individual instructions are categorized by the value of their first byte. Subsequent bytes usually provide addressing offsets or constant data values for use by the instruction. The exceptions to this rule are the *escape* and the *sys* code, which cause the second byte to be evaluated before the instruction type is determined.

The instructions fall into six main categories:

1. Instructions which transfer data between main memory and the evaluation stack which is part of the arithmetic/logic unit
2. Instructions which manipulate data on the stack
3. Instructions which change the sequence of instruction execution by modifying the values of the F and PC registers.
4. Instructions which implement procedure calls and in the process modify registers associated with the management of the stack in main memory.
5. The coroutine transfer which suspends the execution of a coroutine and activates another.
6. Instructions which transfer data between the M-code machine and input/output devices

Appendix 1 gives a descriptions of the instruction set. More will be said about some of the particularly important instructions in later sections.

3.4 The Organization of Programs in Memory

At any given instance, Lilith will be found to have at least ten (or more likely 20) different modules in memory, which may be in use by the coroutines operating within the system. There is a master

Addressing modes:

immediate	n
local	L + n
global	G + n
indirect	S + n
external	FrmTab[m] + n

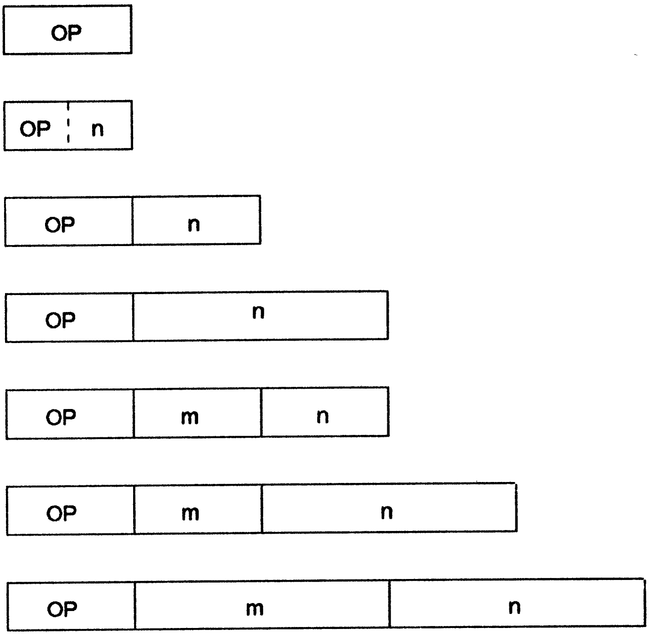


Fig 3.2 Microinstruction Formats

table of addresses for each module which is known as the *module frame table.* The table begins at the absolute address of 20 (in hexadecimal) and has room for 96 entries. The number assigned to a module is used as an index to obtain its address from the table. Modules may reference other modules through the use of the module frame table. The address of the module frame table is not a matter of arbitrary program selection; it is fixed and known by the microprogram of Lilith. The *module frame table* is shown in the upper left hand corner of Figure 3.22

Modules

A *module* is a collection of procedures and data structures for performing a given task or related set of tasks. It also has a structure which is a matter of definition in the microprogram defining the instruction set. This structure consists of a global data frame and a code frame. Modules are normally loaded into memory in contiguous areas from a low address above certain reserved addresses to high addresses. The address of the first word of a module is entered into the *module frame table.* All references to a module from external points use an index value into the *module frame table* to fetch the pointer to the module. This pointer specifies the global data area where, in turn, the first word is another pointer which gives the F register value for the start of the code frame. At the start of the code frame is a second table for addresses of procedures found within the module. There are as many pointers as are necessary for all the procedures of the module. Figure 3.1 shows this relationship.

The Global Data Frame of a Module

As mentioned, the *module frame table* contains entries which point to the beginning of the memory allocated to a module. This pointer value points directly to the memory area reserved to hold the variables declared globally with the module. The first three locations of this area are not assigned to global data variables but are used for special functions. These functions are:

Word 0: This word contains a value which, when placed in the F register, is used by the instruction fetch unit to fetch instructions for execution. This value is not the actual starting address for the code frame because the F register has a 1-bit offset from the normal address so the word actually contains the address divided by two.

Word 1: This address will initially be zero and will be set to a non-zero value with the first call of the main procedure of the module. This word is used as a flag to inhibit multiple initializations of the module.

Word 2: String constants needed by the program are appended to the memory assigned to global variables and referenced indirectly and post-indexed through the pointer found in this location.

After the first three words in the data frame, the next storage locations are allocated for use as global data variables.

Besides being addressable from all procedures of a module, global data variables have three important characteristics which affect their function in a program: First, unlike local variables, the storage allocation and "life" of the global data variable is permanent as long as the program module defining the global variables is resident in memory. In Modula-2, the existence of variables can even extend beyond the completion of the program which invoked the loading of the module containing the variables. This provides a mechanism for passing information from one coroutine to another. Second, global variables are the only variables which may be exported from one module to another. Third, global variables are equally accessible by all coroutines which execute with the module which has defined them or imported them. These features, universal scope and permanence beyond the time boundaries of program executions, extend the use of global variables beyond the normal functions seen in Pascal programs.

Because of these characteristics, global variables can assume a special role in Modula-2 programs which supercedes their use in Pascal programs. This role is related to three different aspects of communications between modules, programs, and coroutines:

First, because of their ability to be exported, global variables are the means of communicating data between separately compiled modules which import and export them. This becomes an important consideration in the management of complex programming problems as the selection of which global variables to export and import significantly affects the efficacy of the programs using them.

A second communication aspect relates to the lifetime of the global data variables which are associated with the resident program modules in memory. In Lilith, it is not necessary to purge a module from memory simply because the task which caused it to be loaded has terminated. It is possible to reuse a loaded module together with all accumlated information in its global variables by using the module with a newly assembled selection of program modules to be used by a new coroutine. The new coroutine thereby benefits from the accumulated information which the previous task computed.

The last communication function of global variables is associated with the nature of modules which are used by more than one coroutine within their overlapping lifetimes. Because global variables are linked with a program and not with a coroutine (as is the case with local variables which are allocated privately and separately to the stack of each coroutine), the global variables become a commonly shared resource to the coroutines which reference the same program module. This can be an important means of communication, but it can also become a source of failure if consideration is not given to the well known problem of synchronization and deadlock.

The Code Frame

As a module is compiled, each procedure encountered in the process is assigned a number in sequence by which it is referenced from other modules and from other procedures within the same module. An address for the entry point of each procedure relative to the address held in the F register is placed in a table at the start of the code frame. The sequence number assigned to the procedure is the index value through which the offset value is found. Therefore, the starting address for the code frame is also the location where the address for procedure 0 of the module is found. The main program section of a module is always *procedure 0.* There will be as many words of procedure addresses as there are procedures within the module.

The code frame is currently assigned by the operating system to the first even word address just above.the global data and the string constants. However, there is no reason it could not be assigned to another memory area. If the amount of storage of Lilith available for data variables becomes a limiting factor, the code frames of each module can still be relocated out of the prime addressing space.

Procedure 0 of a Code Frame

Procedure 0 of the code frame is always the main program procedure of a module. When a given program module makes references to procedures in any external modules, the main program procedures of the external modules are called for execution before the execution of procedure 0 of this program module is completed. In such instances, the main program procedures of the subordinate modules serve as initialization routines. This technique applies to any level of nested module references. However, for the case where a module is referenced by more than one other module, the main procedure of that module is only executed a single time because of special program instructions which test the state of the flag stored in the second word of the data frame as described above.

Addressing of Procedures

In contrast to conventional methods of making a procedure call according to the starting memory address for the procedure code, a procedure call in Lilith is referenced only by a simple number representing the sequential ordering of procedures within the module, i.e. 1, 2, 3..N. This number provides an index value from the start of the code frame to a single memory location which then provides the actual starting address of the procedure as an offset from the code frame. For internal procedure calls within the same module, only the relative procedure number is required. For procedures external to the module where the call is made, the procedure must be called through the use of its 8-bit module number and its 8-bit procedure number.

The addressing of procedures and modules by an index value into a table provides two advantages which may not be readily apparent. First, since all procedure entry addresses are located by index value rather than address, relocation of a code frame to another address is managed quite easily, requiring nothing more than the change of the pointer in the data frame. Second, procedure variables are nothing more than integer table indexes. This also simplifies relocation of code problems and renders the problem of loading and execution of new modules relatively simple.

3.5 The Stack, Procedure Activation Records, and Local Variables

Initially, all available memory not in use for modules is assigned to the stack of the first coroutine. Three registers, S, L, and H, control the use of this memory. The S-register (S meaning "stack pointer") points to the lowest unused storage location or top of the stack. The H-register points to the highest free storage location for purposes of overflow checking, and the L-register is used as a register holding the base address of the most recently created procedure activation record and the local variables of the active procedure. From this address, offsets are computed to memory locations within the stack. As the memory of the stack is allocated for procedure activation records and local variable storage with each procedure call, a comparison is made to determine whether or not the S-register has been incremented beyond the bounds allocated for use by the stack. Each procedure call causes the S-register to be incremented by four plus the number of local variables defined for that procedure. At any given instant, the top memory locations of the stack will hold: a) the local variables used by the procedure currently in execution, and b) the activation record necessary to return the execution control back to the program segment which called the procedure currently in execution. See Figure 3.3.

The Procedure Activation Record

Each time a procedure is called, it is necessary to place enough information on the stack so that the execution of the program making the procedure call can be resumed correctly, and so that the activated procedure may have access to information at lower nesting levels. Four words of information must be stored for this purpose. These words have the following use:

Word 0: If calling a procedure in an external module, then this word contains the pointer value from the module pointer table for the module making the call. If calling a procedure within the same module, then this word contains a pointer value establishing a "static link" to variables on the stack which are considered at a lower nesting level, or global, to the called procedure.

Word 1: This word contains a pointer to the start of the next lower activation record. This is the so-called "dynamic link" which is necessary to deallocate storage from the stack when the current procedure is terminated.

Word 2: This word will contain the PC value for the next M-code instruction following the instruction which called this procedure. If this procedure was called from within the same module, then there is a '0' in the most significant bit of this word. If the call was made from an external module, then the most significant bit of this word is a '1', and the procedure return instruction will know to restore the G-register from word 0 of this record, as well as the F-register from the first word of the global data area pointed to by the G-register.

Word 3: This word is used to save the state of an interrupt mask in the event that the module entered wishes to change it.

The L-register

Each time a procedure is called, the stack is expanded twice. First, as the procedure call is made, a procedure activation record is constructed on the stack, occupying four words of the stack. Then as the procedure is entered, more storage is allocated to accommodate local variables which are declared within the procedure. Parameters of the procedure call will also occupy local variable space. The newly invoked procedure then begins to process information through these memory locations. To facilitate this, the L-register is assigned an address pointing

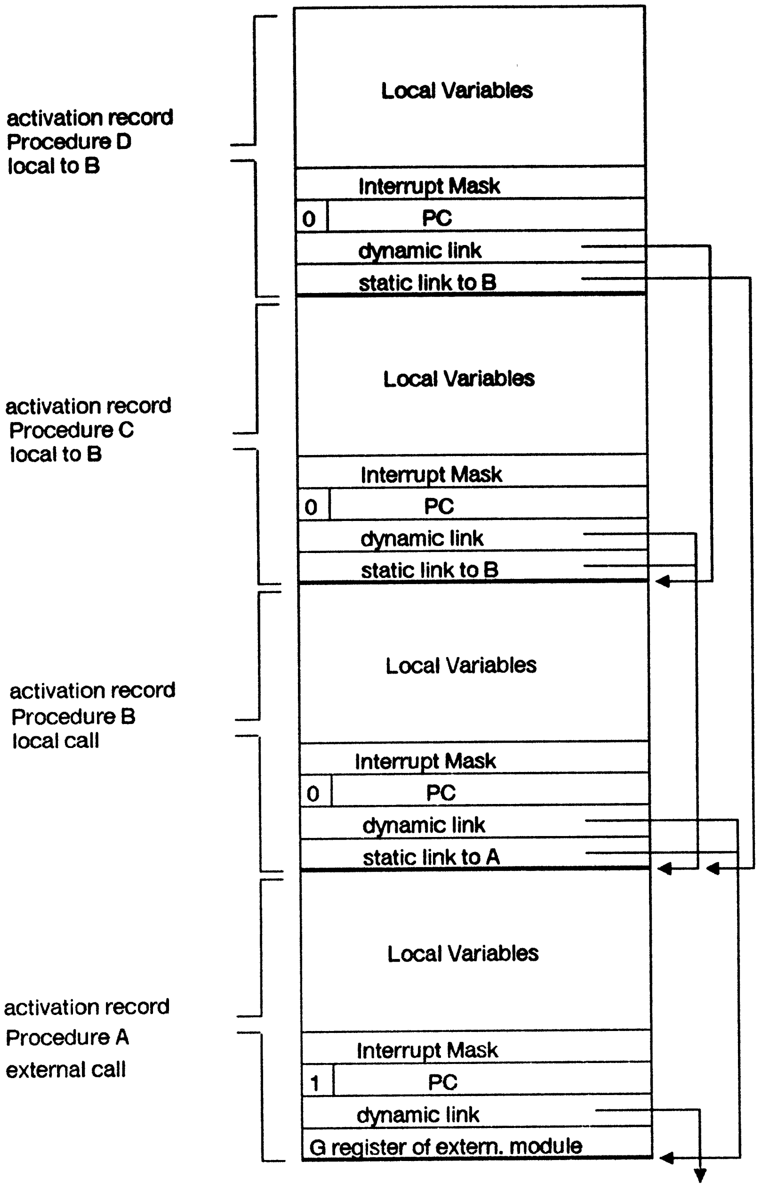


Fig 3.3 Procedure Activation Records in the Stack

to the base of the procedure activation record. The invoked procedure may then address these memory locations efficiently as small offsets from the L-register base. Instructions are available which use either four or eight bits for the offset information. The efficiency of the program object code is improved greatly through this mechanism.

The use of the L-register in this manner implies an additional overhead for the operations of procedure call and return from procedure. Each of these operations must alter the contents of the L-register so that subsequent fetches of local variables will be correctly performed. In the case of the procedure call, it is necessary to save the value of the L-register into the dynamic link (word 1) of the new activation record so that it may be restored later. Then the base address for the new activation record must be loaded. It is held in the S-register at the time of the procedure call. The return from procedure instruction must restore that L-register value in use by the program segment which made the procedure call. This is done by loading the L-register from the dynamic link which was prepared by the procedure call instruction.

3.6 Addressing of Data Variables

The storage addressing schemes of the Lilith virtual machine directly reflect the needs of the Modula-2 programming language. In Modula-2, variables may be allocated in a multitude of ways, although they are referenced through program instructions in one of only four separate ways:

Offset from the G-register:

The G-register points to a data area reserved for use with the module which is currently being used by the active coroutine. Data variables which are considered global to the entire module are allocated from memory in this area. There are special instructions giving access to these variables which minimize the number of bytes necessary to make the reference.

Offset from the L-register:

In a manner analogous to the use of the G-register, there are instructions which operate on data variables which have been allocated locally by a procedure to the stack of the active coroutine. This includes the variables which are declared as formal parameters of the procedure. The method of use again involves the use of special instructions which reduce the number of bytes necessary to make the reference.

Offset from the G-register area of an External Module:

This method of addressing gives a module access to global variables in other modules. The mechanism is similar to references to its own global variables but requires an additional memory cycle to fetch the external module's data frame table entry.

Offset from a Calculated Address

Numerous references to data variables require either references to pointer values stored in other variables or references made with the help of calculated addresses as is necessary for indexed addressing into arrays. In such cases, the needed address are put on top of the expression evaluation stack as the result of other variable fetches and arithmetic operations. To make references with such addresses, the instruction set includes a number of instructions which provide efficient memory addressing based upon the value found in the top of the stack.

As a final comment on addressing, it seems worthwhile to point out once again that data operations on local data variables affect only the local storage of the coroutine using the procedure. On the other hand, data operations affecting global variables affect all coroutines which use the particular module to which they belong.

3.7 The Evaluation Stack

The evaluation stack of Lilith is only 16 levels deep and it does not automatically overflow into the main memory stack as has been the case in other stack machines. Actually, here is no way to check the stack for overflow. This poses an interesting question as to how Lilith avoids undetected errors caused by accidentally overfilling the stack. The answer is contained in a simple rule: All programming in Lilith is done using the Modula-2 compiler. The compiler never uses the stack for any kind of operation where it cannot determine the dynamic level of evaluation stack usage--as in the case of building arguments on the evaluation stack by recursive procedure calls.

This rule implies that each executed statement leaves the evaluation stack in an empty state at the end of its interpretation. Because function calls may occur in the middle of a statement, function procedure calls require that the evaluation stack be saved onto the main memory stack before calling the procedure. A return from function must restore the evaluation stack.

Coroutine transfers must also save the state of the evaluation stack. They do so by popping words from the stack until it is empty. The words taken from the evaluation stack are pushed onto the main memory stack. Finally, the count of words popped from the evaluation stack is pushed onto the main memory stack. The restoration of a coroutine's state reverses this process.

3.8 Coroutine Transfers and Interrupts

An explicitly programmed coroutine transfer has the form:

```
TRANSFER(old,new);
```

where the parameters *old* and *new* are pointers to process descriptors. The process of executing a coroutine transfer begins with saving the contents of the evaluation stack onto the main memory stack of the coroutine being suspended. Then, the entire state of the machine is saved into the process descriptor whose address is found in the P register of the machine. When the entire state has been saved, the address of this process descriptor is saved in the pointer variable "old". Then, the address of another process descriptor is picked up from the variable new and the registers of the machine are restored to their last values for the coroutine resuming execution.

(NOTE: The machine takes the value from the variable "new" at the beginning of the operation and saves it in an unused register internally during the saving of the coroutine state. Therefore, it is possible for "old" and "new" to actually be the same pointer variable.)

Interrupts

A hardware interrupt requires that the state of the machine be saved and that a special driver coroutine be activated to satisfy the source of the interrupt. When the need has been taken care of, the state of the machine must be restored and the interrupted coroutine allowed to continue processing. In Lilith, this operation is identical in execution to the coroutine transfer which can be explicitly programmed. The interrupt handler becomes just another coroutine. The only difference is that instead of allowing *any* process descriptor pointer to be used as a parameter of the coroutine transfer instruction, an interrupt-caused transfer must use assigned locations for the parameters of the transfer operation. It is the responsibility of the programmer to see that the process descriptor pointer for an interrupt handler is stored into the correct memory location which is assigned to that interrupt.

This characteristic of Lilith has some interesting properties. For example, once a coroutine has been written to perform a given function, the programmer can allow that function to be initiated either by a real interrupt or by an explicitly programmed coroutine transfer. The interrupt driver need not know the difference and also does not need to respond differently. This feature of Modula-2 programming is discussed more thoroughly in the Modula-2 reference report by N. Wirth [WIR82].

Master Reset

When the master reset button of the system is pressed, the M-code machine is initialized to a known initial state and begins operation. It first initializes an operating system program which eventually loads a *command interpreter* program that can accept commands from the terminal.

Because Lilith is a developmental machine, it is expected that a programmer will occasionally cause the machine to lock up through faulty programming. In many cases, the the reset button will have been pushed to escape from the unfortunate programming situation. As an aid to the programmer caught in this circumstance, the machine accepts a keyboard command just after reset has been pushed which causes the entire contents of memory to be written to the disk. Then the programmer can discover the reasons for the failure in his program by using a *debugger* to analyze this file.

4 The Lilith Processor

In the previous chapter the Lilith computer was presented as it is seen from the perspective of the compiler writer, or, in other words, as one who prepares programs in M-codes would see the machine. The Lilith processor as seen at that level was called the M-code machine. Now the hardware will be presented at the lowest and most intricate level of operation. At this level of operation, M-codes are not viewed as a single indivisible operations, but rather as a the sequences of *microinstruction* steps which are necessary to perform the function of the M-codes. The collection of all the microinstruction steps necessary to emulate the functions of all M-codes is called the *microprogram*, or the *M-code interpreter.* This program is the vital link between the actual hardware of Lilith and the *virtual* M-code machine. In this chapter, the function of the hardware components will be presented to explain what hardware functions are available for the microprogram to use in emulating M-codes. The microprogram itself is listed in the Lilith Hardware Manual [Ohr82].

4.1 The Microinstruction Cycle

The execution of a microinstruction is the most basic operation in Lilith. It is usually, but not always, completed in 150 nanoseconds. It's period can be made to be longer, but probably not much faster. When attempts were made to make Lilith function with a microinstruction period significantly shorter, incorrect operation resulted because some of the electronics signals did not have sufficient time to propagate through the Lilith circuitry and find a stable value.

The timing of a microinstruction cycle follows a simple protocol. First of all, the start of a microinstruction execution cycle is always signaled by the leading edge of a signal called the *clock*. Shortly after the occurrence of this clock pulse (within several nanoseconds), the microinstruction register outputs find the stable values which are to be the microinstruction commands for the duration of the microinstruction cycle. There are 40 such command bits, each of which has its own meaning and which controls different parts of the processor circuitry. The circuit elements of the processor which are sensing the states of the command bits divide themselves into two categories: those who are commanded to provide some information for use during this microinstruction cycle, and those who are commanded to receive some information during this microcycle. Those circuit elements who are commanded to supply information begin at once to respond to the command bits. Because the data must sometimes flow through a number of circuit elements, it is important that each component be as fast as is required to allow the data to completely propagate throughout the system and stabilize before the end of the 150 nanosecond microinstruction period. Then, at the end of the microinstruction cycle, the leading edge of the clock comes once again, and at this instant, those circuit elements, i.e. registers or flip-flops, who are being commanded to receive data, do so, capturing the stable data at their inputs. The leading edge of the clock which has just signaled the end of one microinstruction cycle is simultaneously the starting signal of another cycle, and so it goes.

It is characteristic of the execution of a microinstruction cycle that at the beginning, all signals in the system are in a state of flux and considered to be invalid. Shortly before the end of the cycle, all signals stabilize to a value (true or false) and can be correctly sampled. The initial period of incorrect data will not yield erroneous results because all circuitry of the system will be synchronized to the leading edge of the clock and thus insensitive to any false data except if it were to occur just before the edge of the clock. This is what is commonly known as the technique of synchronous design. (See fig 4.1)

4.2 The Microinstruction Register (MIR)

The microinstruction register consists of 40 flipflops which are loaded with command bits at the beginning of each microinstruction cycle. Each of these command bits has a separate meaning assigned to it and circuitry connected to it which responds when the command bits assume the value for which the meaning is defined. Some of the bits have more than one meaning. This is accomplished by having multiple command formats and using two of the 40 bits to select which

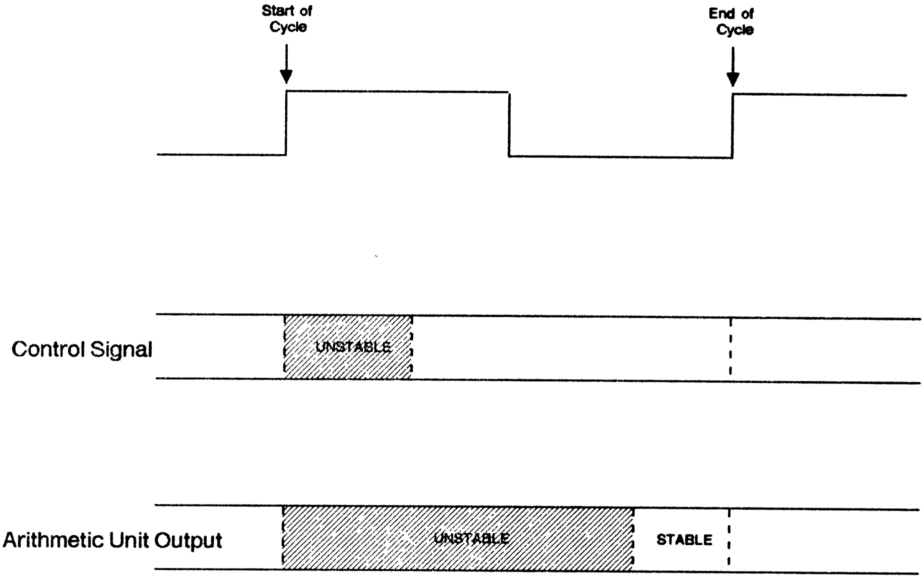


Fig 4.1 Microinstruction Timing

format is operating. In such cases where command bits have different meanings in different formats (not always the case), the circuitry responding to the command bits must also be capable responding to the format selection.

Figure 4.3 shows the three microinstruction formats in Lilith.

In formats 1 and 2, the majority of the bits are used to control the ALU. In format 3, no bits at all are used to control the ALU. It is totally disabled during format 3 instructions.

In format 1, 8 bits are used to control the source and destination selection of the BUS. In format 2 these bits are used to provide a constant to the bus. The destination in this case is always the ALU.

In all three formats, there are bits controlling the selection of an address to use for the next microinstruction.

4.3 The System BUS

The main participant in the execution of a microinstruction is the system bus (BUS). It is the highway by which data travels from one end of Lilith to the other. It is 16 bits wide and operated by *tri-state* logic. A typical microinstruction cycle using the bus might cause the following to occur:

1. A register in the AMD2901 is commanded to give its contents to the arithmetic adding circuit inside of the AMD2901 bit slice chips.
2. The AMD2901 arithmetic circuitry is commanded to give its data to the chip outputs
3. The ALU board is commanded to pass the AMD2901 output data to the BUS.
4. A register in the memory system is commanded to receive the data from the BUS as an address to be used in a memory cycle.

The connection of the system BUS to other elements in the system is shown if figure 2.1.

The control of the system BUS comes directly from the microinstruction register. In format 1 microinstructions, 4 bits (as shown in figure 4.n) are allocated to select the source of data on the bus, and 4 more bits are allocated to select the destination on the bus. In format 2 microinstructions, the 8 bits a gated directly to the BUS as a constant, and the destination for the BUS data is the arithemetic/logic unit. (See fig. 5.4)

4.4 The Control Store

The control store of Lilith can have as many as 4096 instructions stored in banks of bipolar *read-only* memories (ROMs). Typically, only 2048 are used. During the execution of each microinstruction, the control store is fetching the microinstruction to be used in the next cycle. Since a microinstruction cycle is only 150 nanoseconds long at times, it is necessary to have ROMs which are quite fast. Also, since there are no additional timing pulses which can be generated, it is necessary to have ROMs which supply information continuously based only upon the address given to them.

4.5 The Microinstruction Address Circuitry

A large amount of circuitry on the MCU board is dedicated to the function of generating the address for the control store ROMs. This address can be generated in several different ways:

1. The previously used address can be incremented. This occurs most commonly.
2. The microinstruction can be of type 3 and may specify directly a 12 bit address. The address may be specified either conditionally or unconditionally. Also, a jump subroutine operation may be coupled with the direct transfer which causes the saving of a return address on a four level stack.

3. The return address of a subroutine call can be taken from the four level stack.

4. A 12 bit address can be taken from the output of a special map ROM which uses as its address the least significant eight bits of the system BUS. The intended use here is to accompany this mode of usage with a command to the IFU to place the next M-code onto the BUS. Therefore, a direct jump takes place to the next M-code microinstruction sequence

5. An interrupt vector address can be given which is equivalent to the number of the interrupt plus eight. This vector is given by the hardware in a cycle when the microinstruction is trying to decode a new M-code. The vector displaces the M-code jump address by disabling the jump map ROMs.

The heart of the circuitry for the microinstruction address generation is based upon three AMD2911 sequencer chips. These chips each process four bits of the 12 bit address, and their outputs go directly to the control store ROMS. Internally, these chips have an incrementer circuit which automatically increments the address used in each microinstruction. The incremented address is then saved in a register within the chip so that it may be selected for use during the next microinstruction cycle. There is also a four level subroutine return address stack within the chip which will save the incremented address from the previous microinstruction if the current microinstruction is performing a jump subroutine instead of using the incremented address. Whenever an address is used which is neither the incremented version of the previously used address nor the return address of a subroutine, it must be generated externally and passed into the AMD2911s through the direct inputs which the chips offer.

Figure 4.2 shows a simplified block diagram of the structure which generates the microinstruction address.

4.6 The Clock Generation Circuitry

As has been indicated previously, most events which occur in the machine are synchronized to the leading edge of the clock signal. For the greatest percentage of microinstruction cycles, this signal is generated at regular 150 nanosecond intervals. However, there are some conditions which require delaying the clock for some period of time, and so the clock generation circuitry is not as trivial as one might expect. The conditions which require the clock to be delayed are as follows;

1. The slow bit in the microinstruction is set. This typically occurs when a decode of an M-code is in progress. This requires more than 150 nanoseconds to accomplish because of the time required to look up the starting address for a M-code's microinstruction execution sequence. The slow bit causes the clock cycle to be extended to 225 nanoseconds.

2. The microinstruction attempts to transfer data with one of the two memory ports, and that port is still occupied with another operation. In this case, the clock is simply stopped until the port is free to be used again.

3. The microinstruction attempts to transfer data with an I/O device which is momentarily occupied. Again, the clock is simply stopped until the device is ready.

4. Lilith is being operated in diagnostic mode and the diagnostic processor unit (DPU) has halted the processor. The clock will be delayed until the DPU releases it by allowing the processor to return to run mode, or by giving a command for a single clock cycle.

5. The master reset switch has been pressed. The clock will not operate until the master reset signal returns to its normal operating condition.

The clock generation circuitry operates from a master oscillator which oscillates at twice the maximum operating frequency of the system (13.333 megahertz). There are actually two clocks which are generated having the same phase and timing relationships. One is the MCU CLK which times all operations on the microcontroller board. The other is the CPU CLK which

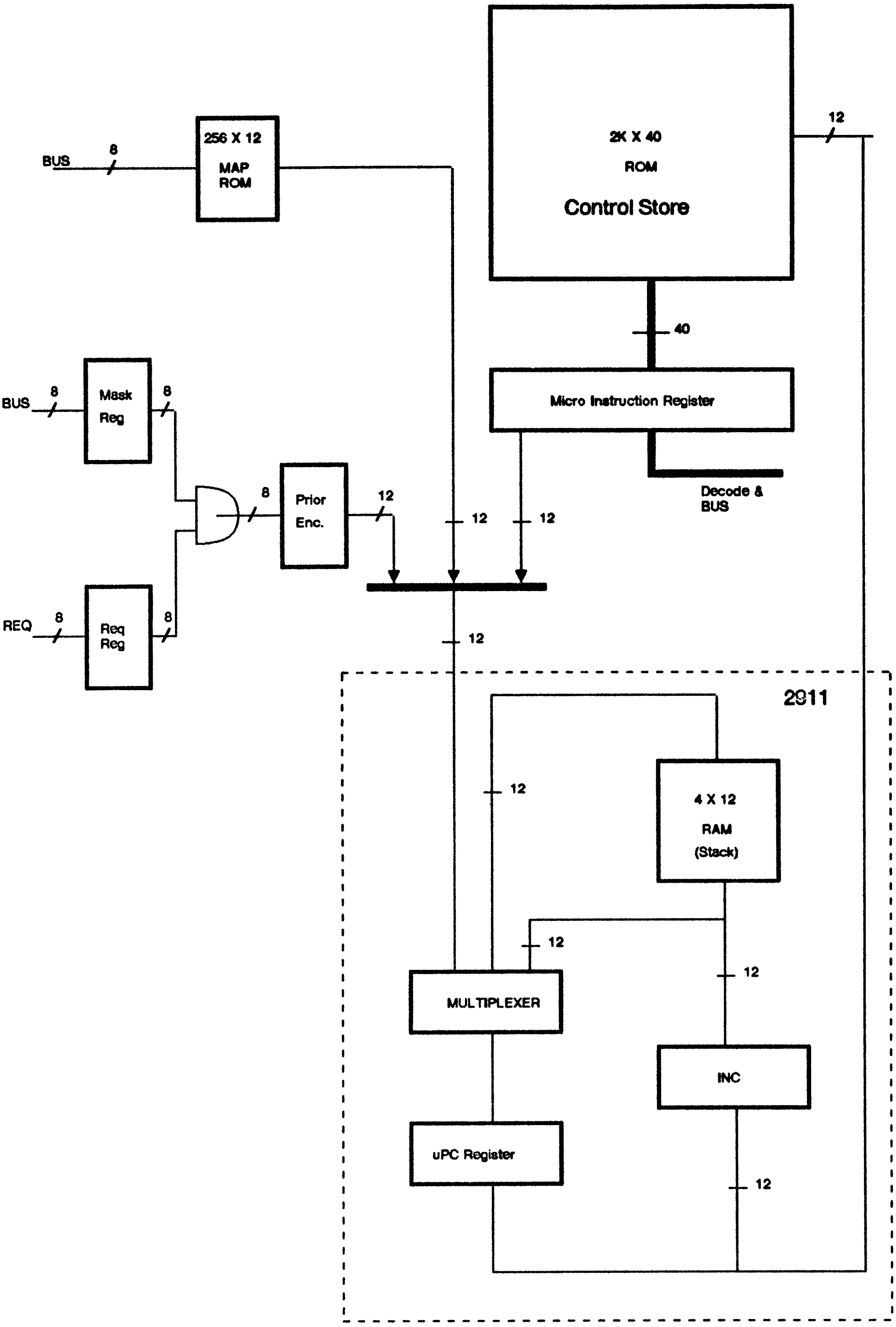


Fig 4.2 The Micro-Control Unit

connects to all other elements of the system. The CPU CLK does not function during the execution of format 3 microinstructions. This prevents the other parts of the system from responding to the alternate use of the bits in the microinstruction during a format 3 instruction.

4.7 The Arithmetic/Logic Unit (ALU)

The work of information processing is done in the ALU. All other elements of the system exist for the purpose of getting information into and out of the ALU. Other elements of the system can store information or transmit it, but this is the one place where data is added, subtracted, ored, anded, shifted, incremented, or inverted.

The heart of this activity is the AMD2901 bit slice processor chip. There are four of them in the Lilith arithmetic/logic unit. This gives it the capability to operate upon 16 bits of data in parallel. The chips provide 16 fast registers which are used as temporary storage locations for data which the ALU operates on. It is possible to simultaneously read two registers of the 16 for use as operands in the arithmetic and logical operations. One of the two registers can also be used as the destination for the operation result. The operations which the AMD2901 provides are simple integer arithmetic and boolean logic operations. All operations are dyadic, but the input circuitry offers the ability to select zero as one operand so that monadic operations such as increment and complement are possible. There is also a special register in the chip called the Q register which has been configured to be useful for the purpose of performing multiply and divide operations.

The capabilities of the 2901 were augmented through the addition of a barrel shifter and a 16 level stack. Both of these circuits connect to the direct input port of the 2901 with tri-state logic. They cannot be used simultaneously. The barrel shifter has been constructed so that it not only performs a barrel shift, but it can also create masking constants by shifting in 1's in from the left.

The 16 level stack is the physical implementation of the arithmetic evaluation stack. It is constructed from fast registers and operates at the maximum operating speed of the machine. It is possible to perform all stack operations in a single microinstruction cycle, i.e push, pop, add, etc.

The results of all arithmetic and logic operations are sampled at the end of each microinstruction cycle into the contents of a *condition code* register. Then, the condition codes must be tested immediately following each instruction to detect error conditions. The interpretation of which condition codes represent an error depends, of course, on what type of M-code is being interpreted. i.e. overflow for cardinal arithmetic is sensed differently than overflow for integer arithmetic.

Figure 4.3 shows a block diagram of the ALU.

4.8 The Data Port

Information from the main memory of Lilith travels to and from the arithmetic/logic unit by way of the system BUS. A special memory port interface exists in Lilith just for the purpose of handling the transfer of the contents of memory cells which will be used only for data operations. Another interface handles the transfer of memory information which is used as M-codes. The memory port which is used for data transfers is called the *data port.*

A read operation is initiated when a microinstruction selects the *memory address register (MAR)* as the recipient of data on the bus. This will be treated as the least significant 16 bits of an 18 bit address. If no data was transferred previous to this microinstruction to the *high memory address register (HMAR),* then the high order address bits will be set to zero by default. After the memory address register has been loaded, the memory subsystem will perform a read operation on the addressed memory location. If there is no interference from other memory requests, the data will be available in about 500 nanoseconds. The data from that memory location can then be brought onto the system BUS by specifying the *memory data register (MDR)* as the source of a microinstruction.

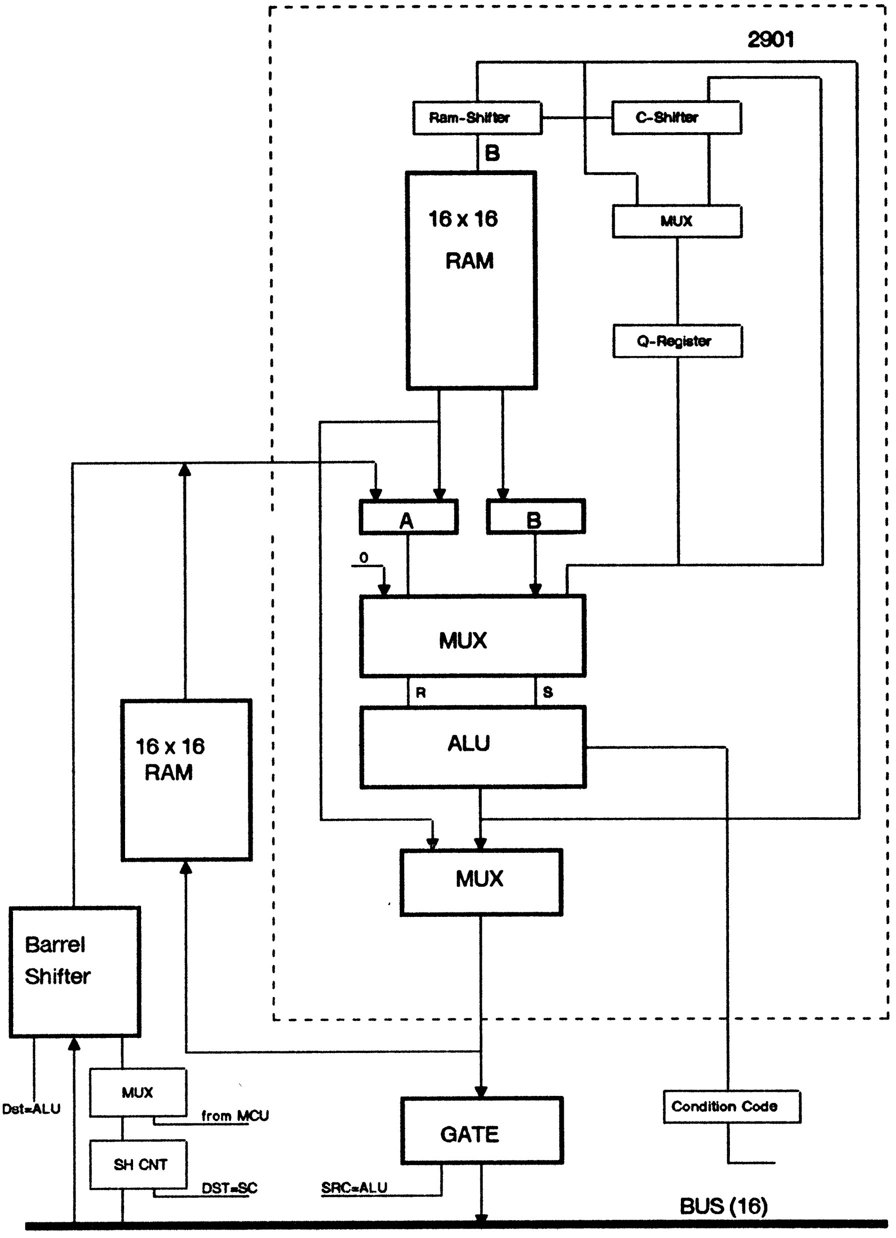


Fig 4.3 The Arithmetic-Logic Unit

For a memory write operation the protocol is identical for the transfer of the address. The processor must only transfer the data for writing to the memory data register *before* transferring the address. By transferring the data to the memory port before the transferring the address, the port is able to distinguish between a write and a read operation.

4.9 The Instruction Fetch Unit (IFU)

The instruction fetch unit resolves the problem of addressing memory by bytes when the machine is fundamentally a word (16 bits) oriented device. In the previous chapter, it was explained how the high level language orientation of the machine allowed the architecture of Lilith to function without the capability to perform any direct program addressing, i.e. all instruction addressing is either by relative offset or by table look-up. This characteristic has allowed the instruction fetch unit to be cleverly constructed from two registers: the code frame register (F) and the program counter offset register (PC). The contents of these two registers are directly managed by Lilith instruction addressing techniques. Whenever a procedure is called in an external module, the correct value for the code frame register is found as the first word of the global data frame and the new value for the program counter offset register is found in the table of procedures at the start of the code frame. Whenever an M-code commands a relative jump, the processor adds the relative amount to the contents of the program counter offset register and returns it to the instruction fetch unit. Inside of the instruction fetch unit, a simple adder circuit combines the contents of the two registers into an address which is given to the memory whenever M-codes must be fetched. After each M-code is fetched, the count in the program counter offset register is incremented.

Transparent to the rest of the processor is the fact that the IFU actually fetches eight bytes of instruction from memory each time its data buffers are empty or are rendered obsolete by the execution of a program branch. Oblivious to the changes in the program counter, or the requests for memory cycles, whenever a M-code is needed, the processor simply requests the next sequential M-code by addressing the instruction fetch unit as the BUS source. If the processor happens to request the last M-code of a group of eight, the IFU immediately requests a new memory cycle to refill the buffer.

Figure 5.8 shows a block diagram of the instruction fetch unit.

4.10 The Memory

The Lilith memory can be shared by as many as eight subsystems of the machine. The memory cycles are allocated upon request according to a priority established by the circuit which senses the request. The address and data lines running to the memory are tri-state. When a memory cycle is granted to a subsystem, a selection signal tells the subsystem that it can drive its signals onto the memory's address and data bus. During the course of a memory cycle, timing signals which emanate from the memory system's timing control tell the subsystem when to sample data from the system, and when the cyle is complete.

For read operations, the memory circuits are connected so that 64 bits may be read in parallel from a single address. These 64 are logically four 16 bit words of memory. The two least significant address bits are not used in the read operation unless the port desires to receive only a 16 bit memory word. In that case, the two least significant address bits control the operation of a multiplexor circuit which selects a 16 bit word from the four which the memory has delivered.

For write operations, the 64 bit data path does not exist. There are only 16 data bits which are used. Each bit is wired to four separate memory circuits, but only one of the four memory circuits is activated to accept the data. Here the two least significant bits are necessary. They are decoded to select the word of memory which will be written.

There are two cards in the basic memory configuration. Each has 32 memory chips on it. To enlarge the memory, an additional set of cards is wired in parallel with the basic set. Each set of cards is equipped with bank selection logic which causes the cards to be activated for different address ranges.

5 Decisions Made in the Design of Lilith

The design of Lilith will be presented here as a chronology of the major decisions which were made the design process. Evaluation as to the final merit of these decisions will be deferred to chapters 6 and 7. This chapter will attempt to document the considerations which went into making these decisions.

5.1 Choice of the Processor Technology and the Arithmetic Logic Unit

Our first concern was the choice of a processor technology. We were somewhat limited in our choices because of our conviction that conventional processors were poorly designed for use by high-level languages. We thus eliminated all microprocessors and "computers on a chip" because their architectures were already frozen, leaving us the choice of constructing our own processor out of either bipolar TTL or ECL logic. Our concern was to demonstrate improved performance by better design and not by building the fastest possible machine without regard of cost, we decided to use bipolar Schottky TTL logic. The selection was made because of its economy and wide availability. Consequently, our search was then narrowed to the available bipolar Schottky circuits.

One fundamental constraint was the need to be able to do rapid shifting and masking operations. We anticipated the need for such operations in our bitmap screen graphic functions. We were also concerned about the number of working registers which would be available in the arithmetic logic unit and the flexibility with which these registers might be used. We considered using the Advanced Micro Devices 2903 [Am80], but at the time it was not readily available. This was disappointing to us because the extra registers which it offered were tempting. We also considered the Texas Instruments 481 [TI] but found it somewhat limited in its capability. After some deliberation we concluded that an earlier model bit-slice arithmetic unit from Advanced Micro Devices, the AMD2901, did indeed have adequate power and availability to meet our needs. See Figure 4.3.

The choice of the 2901 as the ALU for Lilith presupposed the usability of the device for many relatively unusual operations required in Lilith's role as a workstation processor. Some of the operations anticipated were for the manipulation of Lilith's high resolution bitmap display. We envisioned the implementation of characteristic bitmap operations such as the movement of an image from one portion of the bitmap to another. In such an operation, the following sequence would be undertaken: a word *s*, being part of the image, would be moved to another area of the bitmap where it would be aligned so that part of it must fit in word *d* and the rest in word *d + 1*. In order to perform the operation, the bits of word *s* must be rotated to align them properly with the bit positions in words *d* and *d + 1*. Masks consisting of strings of 1s and 0s must be generated and used to separate the portions of the words from each other which would be used to construct the final image. Logical AND operations between the mask and the bitmap data are used to remove the undesired portions. Finally, the portions of words *d* and *d + 1* must be logically ORed with the appropriate portions of word *s* to create the desired concatenation of data.

For a "normal" processor, the above operation would require a time-consuming sequence of instructions to perform the shifting, masking and merging operations. By providing correctly designed hardware, we hoped in Lilith to improve the performance in this area. One of the special hardware components added was a *barrel shifter* circuit, a circuit for rotating a 16 bit word to any position with the least significant bit rotating into the most significant bit position. The barrel shifter was constructed from medium scale integration parts especially designed to perform this function. One fortunate characteristic of this design was the possiblity to add OR circuits which would give the shifter the capability to generate the masks previously described. This feature is explained in greater detail in the Lilith Hardware Manual [Ohr].

Another feature which we anticipated to add to the 2901 was a special circuit for the expression evaluation stack. This feature will be discussed in a later section.

5.2 Design of Microinstruction Control Unit

Anticipating that an elaborate microprogram would be necessary to emulate the stack architecture and especially needed for the complex bitmap graphic operations, our next concern was the design of an efficient microinstruction control unit (MCU).

As we analyzed the function which the MCU would be required to perform, it was evident that three basic operations needed to occur in each cycle of MCU execution:

1. the execution by Lilith of the operations specificed by the current microinstruction,
2. the selection of an address for the next microinstruction based upon the results of operations and commands of the current microinstruction,
3. the fetch of the next microinstruction from the control store using the selected microinstruction address.

We knew that each of these operations would involve a significant but unknown signal propagation delay in the circuitry which performed the function. The maximum accumulation of these signal propagation delays would determine the minimum required time for each microinstruction execution. A pictorial representation of these time relationships of these signal delays is shown in figure 5.1.

There are widely known techniques [Mick] for reducing signal propagation times. In one of these, registers loaded by the system clock are inserted in the the information paths of the processor. The effect of inserting such a register is to break the signal propagation path and separate it into two shorter cycles. The total time to execute a given microinstruction is actually increased by such an action because two cycles are required to complete the microinstruction. However, if the location of the register is properly chosen, the execution of two microinstructions can be overlapped, and the net throughput is increased. A general discussion of the techniques for design of such *pipeline* registers is given in chapter 2 of "Bit-slice Microprogrammed Design" by John Mick and Jim Brick [Mick].

For our design, we chose to place a register at the outputs of the control store, called the *microinstruction register* (MIR). This register is used to hold the bits of a microinstruction for the period of one microinstruction cycle. Another register is placed at the outputs of the condition codes, called the *condition code register*. The condition code register captures the results of a microinstruction execution and holds them for testing during the next microinstruction cycle. The effect of the condition code register is to reduce the maximum signal propagation delay so that a faster microinstruction execution rate can be supported.

To understand how this works, consider first how it would work without the condition code register. The execution phase begins immediately after a microinstruction has been loaded into the MIR. If the condition codes were not to be captured into a register, the address generation logic would need to wait for the completion of the ALU operations to begin its operation. It must wait for the condition codes to be stabilized for use in generating the address for the next microinstruction. (Specifications in the 2901 documentation [AMD80] and, later, our own measurements indicate this delay would be longer than 100 nanoseconds.) Consequently, the minimum microinstruction cycle time which can be used is the sum of all the delays. (shown in figure 5.1) On the other hand, if a condition code register is used, then the values of the condition codes are immediately stable and the generation of a new address can begin immediately simultaneous with the operation of the ALU. The minimum cycle time in this case is the shorter of the time required for ALU operation and the combined time required for address generation followed by control store access. (shown if figure 5.2)

By use of the condition code register, the logic to generate address can function immediately, but it can only test the results of the previous microinstruction (and not the results of the current instruction!).

The key feature of this organization is that all microinstructions can be executed faster, even

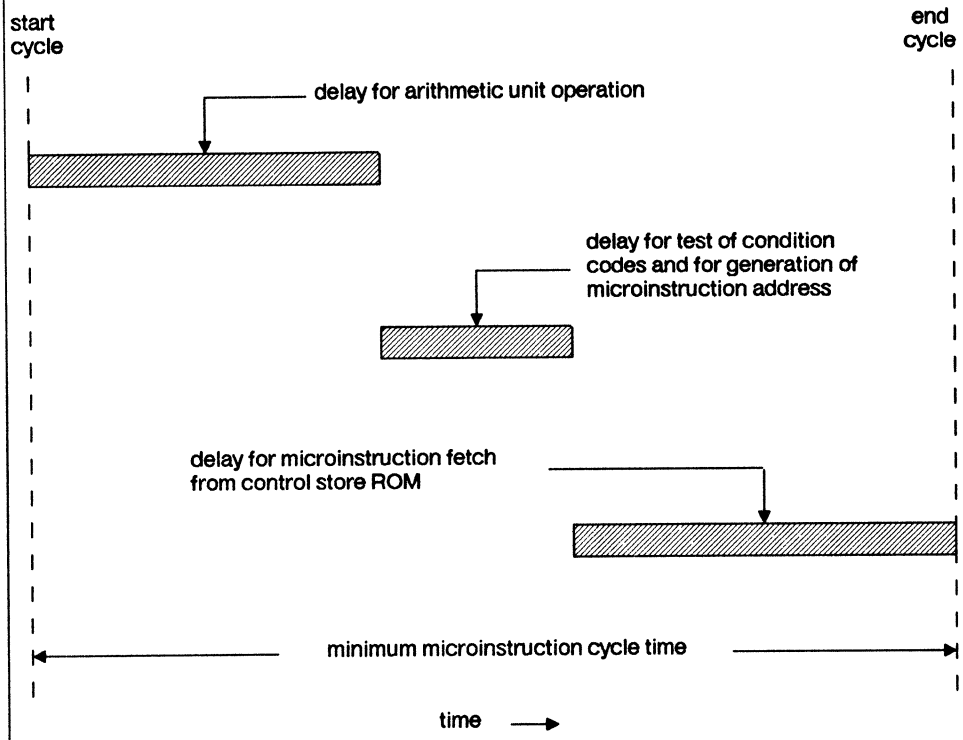

start
cycle
end
cycle
delay for arithmetic unit operation
delay for test of condition codes and for generation of microinstruction address
delay for microinstruction fetch from control store ROM
minimum microinstruction cycle time
time

Figure 5.1 Timing Relationships for a Non-pipelined Processor

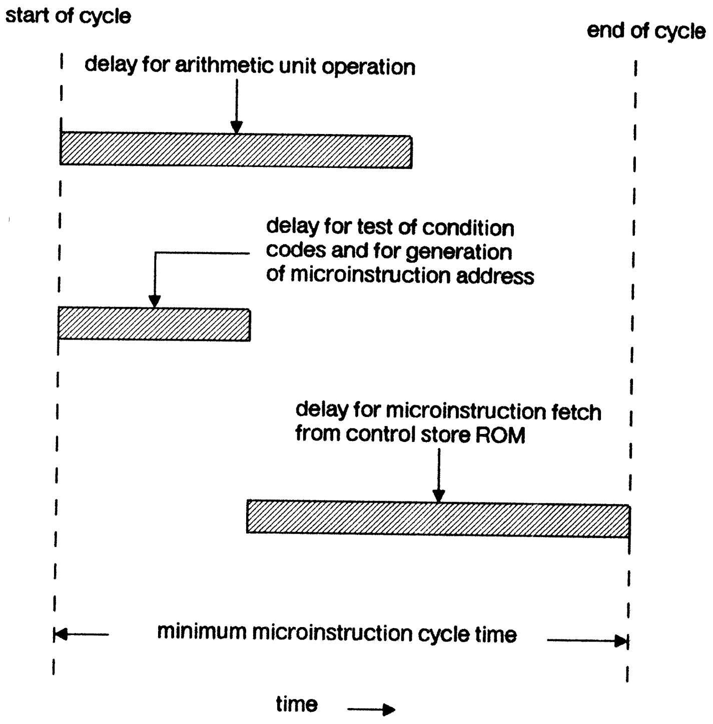

start of cycle
end of cycle
delay for arithmetic unit operation
delay for test of condition codes and for generation of microinstruction address
delay for microinstruction fetch from control store ROM
minimum microinstruction cycle time
time

Figure 5.2 Timing Relationships of a Pipelined Processor

though two microinstructions are needed to perform an operation and check the validity of the results. For example, in the case of the *add* M-code, for example, two shorter microinstructions are needed to perform the operation and then check for arithmetic overflow instead of a single longer microinstruction which would have sufficed if condition codes of a microinstruction execution could have been checked within the same cycle.

The deciding questions, of course, are: how often would such test instructions need to occur, and how much shorter can the microinstruction cycle time be made? Lacking any concrete data as to the dynamic usage of test instructions in the execution of the microprogram (it was not yet written), we used our instincts and chose to include the result register for the condition codes. The merit of this decision will be discussed in the next chapter.

An approximate time relationship of the signal propagation delays is illustrated in Figure 5.2. The overlapped execution of microinstructions is diagrammed in Figure 5.3 in a qualitative fashion.

Design of Next Microinstruction Address Generation Logic

The function of the next microinstruction address generation logic is to provide the microprogram adequate capability to branch to different segments of the microprogram according to the state of the machine and the way the microprogram has been written. There are six conditions which can arise in the execution of the microprogram each of which require a different derivation of the next microinstruction address. They are:

(a) a simple "next" instruction taking as an address the previous address plus one

(b) a jump address, conditional or non-conditional, where the microinstruction address is taken from within the previous microinstruction word

(c) a "map jump" address where the address of the next microinstruction is determined by accessing special map ROM driven from the value of an M-code

(d) a jump subroutine instruction where the current address is saved incremented by one in a special register for use later as a return address, and the subroutine jump address is taken from the microinstruction word

(e) a "return from subroutine" address where the address comes from the register where it was saved during the above mentioned "jump to subroutine" microinstruction

(f) an interrupt vector which is a "hardwired" address provided in lieu of the 'map jump' address whenever an interrupt occurs

None of the integrated circuits available at the time offered the capability to implement all of these features alone, but when used together with some external logic the AMD2911 microcontroller sequencer chip from Advanced Micro Devices offered the capability to implement all of these features. Specifically, it offered the basic mechanisms for sequentially delivering the monotonically increasing sequence of addresses for normal microinstruction access and the mechanism for accepting new addresses directly for jump operations. It also offered the capability to implement a subroutine call by saving a return address in a stack and branching to a new location. This allowed up to four nested subroutine calls for the *jump to subroutine* operations.

After choosing the AMD2911 sequencer, we were left with the problem of adapting it for the execution of the table look-up operation associated with the decode of M-codes, and also with the execution of interrupt operations caused by requests from input/output devices.

The design of the table look-up circuitry for M-codes was relatively simple. It was only necessary to add a small (256 x 12 bit) read-only memory (map ROM). This ROM received its address from the system BUS. By properly selecting the instruction fetch unit as the source for the BUS, the correct M-code is delivered to the ROM which in turn gives as its output, the address of the microinstruction sequence which performs the M-code execution. A special selecltion code in the microinstruction format portion which controls the 2911 allows the address to be taken directly by the 2911.

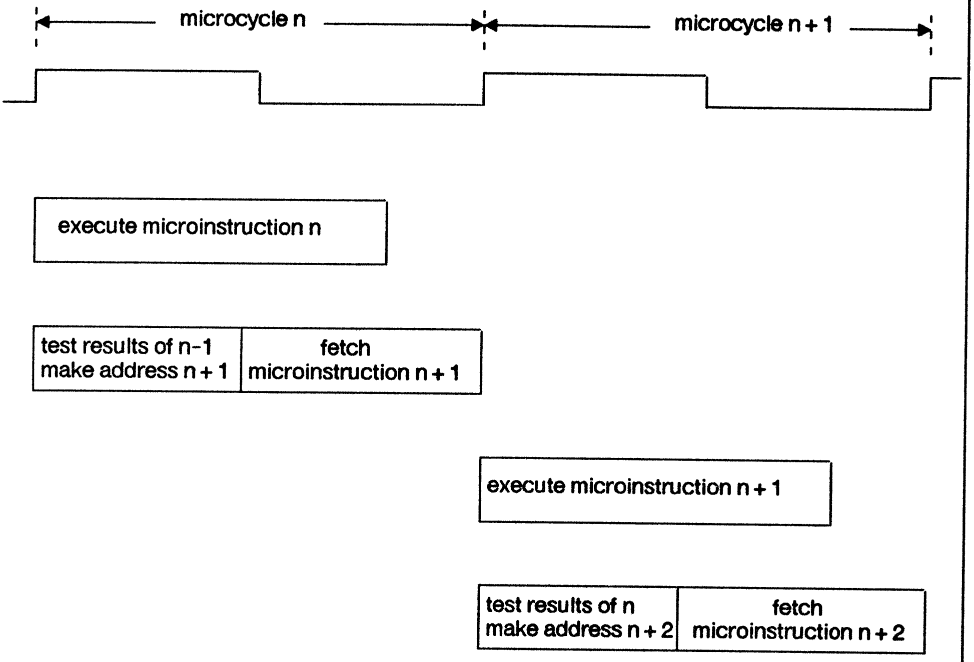


Figure 5.3 Overlapped Execution of Microinstructions

The design of the interrupt mechanism was more difficult. At first we considered implementing an interrupt mechanism which would allow interrupts to occur at any point in the microprogram as had been done in the Alto project [TMLSB]. However, the difficulties associated with managing the use of registers and the proper saving of state led us to give up this notion. We finally developed a technique which allows interrupts to occur only when an M-code table look-up operation is underway. At this juncture, the state of the machine is in a known condition and the operation could be simplified considerably.

The implementation of the interrupt vector in this manner was simple. It required an interrupt request to disable the map ROM during the execution of a M-code decode operation and to enable the circuitry which generates the interrupt vector address. Figure 4.2 gives a block diagram of this circuitry.

The interrupt vector described above was actually added to the design after we had worked out the design of the instruction fetch unit. Initially, our selection of the 2911 was based upon its capability to satisfy the first five requirements.

Design of Microinstruction Format

In choosing the microinstruction format, we allocated the bits according to the needs of the devices being controlled by the microinstruction word. Twenty-five bits were directly assigned to the control of the ALU--the element of the computer most directly controlled by the MCU.

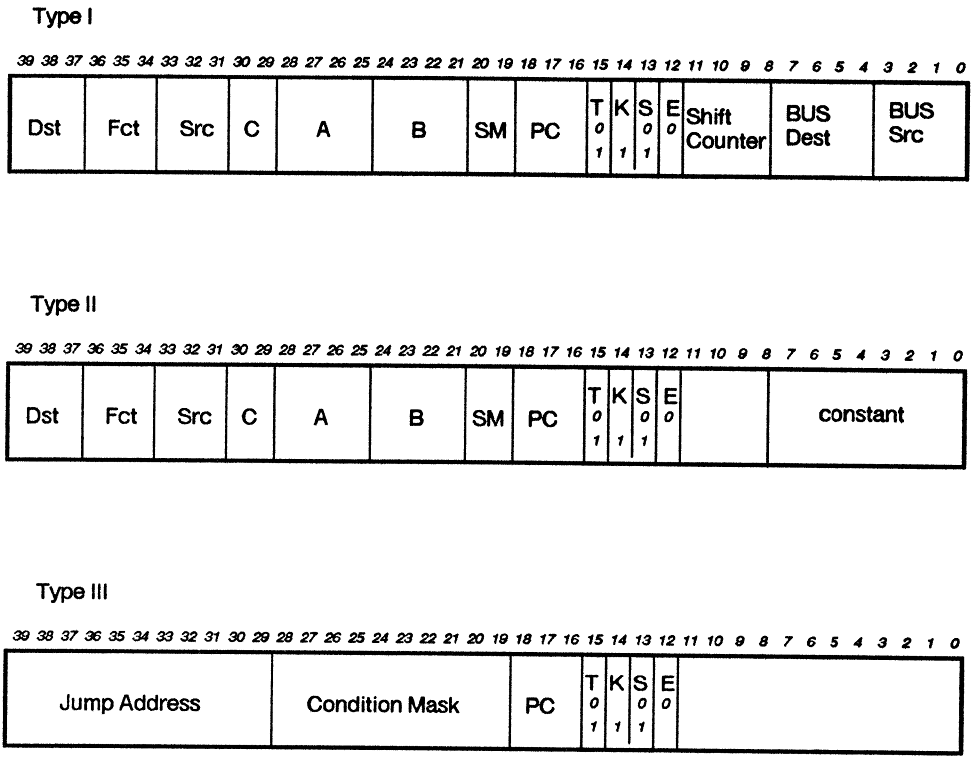
Three bits were allocated to the control of the AMD2911 sequencer, which were used to control the next microinstruction address generation logic.

Eight additional bits were allocated for control of the system BUS: four bits specifying the source, and four bits specifying the destination for information being transferred during any one microcycle. The ALU and the MCU are the only functional units of the computer which are directly controlled by the microinstruction word. The other elements, i.e. Instruction Fetch Unit, Memory Data Port, and the I/O devices are controlled by virtue of being selected as a bus source or a bus destination.

Having made the decision to include a register for the condition codes coming from the ALU, it was questionable whether or not many instances would present themselves which could utilize both a *test and jump* operation and a *compute* operation during the same microinstruction. We felt that this was unlikely, so we availed ourselves of an opportunity to save the use of extra bits in the microinstruction word by establishing two formats: one for *test and jump* operations and one for arithmetic logic operations. The *test and jump* microinstructions use for condition code selectors the same positions of the microinstruction word which are normally used to control the ALU. In order to prevent the ALU from incorrectly responding to the different use of these control bits, the system clock to the ALU is inhibited during *test and jump* instructions. Figure 5.4 gives the format of the microinstruction word.

Generation of Constants

We struggled for some time over the manner in which we would generate constants for use in the microcode. Constants are typically used for masking, as increments to other values, or as I/O addresses. We considered including a special ROM in the ALU for this purpose, but eventually decided to obtain multiple use of the microinstruction word BUS source and destination fields by allowing them to be either used as the control for the BUS sources as normally intended, or to be gated onto the BUS directly as data. This required the use of another bit in the microinstruction word to specify which of the two functions would be active. It also required some special logic to override the normal BUS source and destination selection. When the constant generation format of a microinstruction is selected, the BUS source is automatically selected as the constant in the MIR, and the BUS destination is automatically selected as the ALU. This adaptation to the microinstruction format is also shown in Figure 5.4.



Dst,Fct,Src,A,B	2901 Control fields
SM	Shift mode
PC	2911 Control fields
Sc	Shift counter
T	Instruction time: 0 = long, 1 = short
K(MIR 14)	1 = type 1 instruction, 0 = type 2 instruction
S	Stack enable
E	1 = type 1; 0 = type 2, instruction
C	Carry control for ALU

Figure 5.4 Microinstruction Format

5.3 Expression Evaluation Stack

Stack machines had been frequently discredited as being less efficient than register based machines because of the excessive number of memory operations associated with the manipulation of variables on a stack which is kept in main memory. Consequently, it became a primary objective to discover a mechanism whereby a bipolar stack could be added to the arithmetic unit without undue complexity.

Two issues affecting the complexity were (1) the depth of the stack and (2) the question of how to handle the stack underflow and overflow conditions. Other successful stack machine computers were known to have fast register stacks of relatively small depth which automatically overflowed into main memory [HP] [Bur]. We liked the idea of a fast expression evaluation stack but felt that the implementation of hardware for automatic underflow and overflow into main memory would be too expensive. Three possible solutions were considered:

First, it was thought possible to test the stack for overflow and underflow after each stack operation with the necessary adjustments handled upon detection by microcode subroutines. Then it was realized that the totally pervasive use of the stack throughout the microprogram would require far too many microprogrammed tests for overflow and underflow with subsequent deleterious effects to the processing speed and the size of the microprogram. We rejected that idea.

The second alternative was to include special hardware for the detection of stack overflow and underflow. In this case an interrupt would have to be generated to handle the condition. However, this implied allowing interrupts to occur after each microinstruction, a mechanism we had decided not to include in the hardware because of its complexity. So, we looked for an alternate solution which did not burden the machine with the overhead of managing the stack in this fashion.

Finally, realizing that we were prepared to constrain the programming of Lilith totally to the use of a high-level language compiler, it was concluded that a limited use of the bipolar stack would be possible without implementing overflow into the main memory. This could be accomplished if the stack were used only for expression evaluation and other operations where the depth of the stack usage could be controlled by the compiler. The consequences of such a restriction were that some potential uses of the stack were not possible. As an example, it precluded leaving any data on the stack while executing a function call. Since functions can be called recursively, the compiler would have no possibility to test the level of stack usage. In order to solve this problem, special instructions were included to allow the expression evaluation stack to be saved to the main memory stack before a function call and restored after a return. We were also able to use this operation whenever a coroutine transfer took place.

For its actual implementation, the stack was constructed from standard bipolar memories. The stack addressing mechanism was developed from a synchronous up/down counter and a four-bit adder circuit. The normal state of the stack address counter was to address the next empty register of the stack (for push instructions). For *pop* operations, the adder circuit is used to instantaneously and temporarily decrement the stack address to the last used location. At the end of the microcycle, the stack counter "catches up" by decrementing its value at the end of the cycle. [Figure 5.5]

The connection of the bipolar stack to the 2901 bit-slice chips proved to be a uniquely compatible combination. The effectiveness of the connection satisfied every need. Because of an internal path in the 2901 allowing data from the register banks to bypass the internal adder circuitry in the 2901, it was possible to configure microinstructions which could perform all stack operations in a single 150 microsecond microcycle, i.e. *push, pop* and basic operations between the top levels of the stack, such as *add, subtract,* or *compare*. Figure 5.6 shows how this was possible.

After we had discovered that the 2901 was an ideal host to the addition of such a stack, we considered increasing the size of the stack beyond sixteen levels; however, we could find no

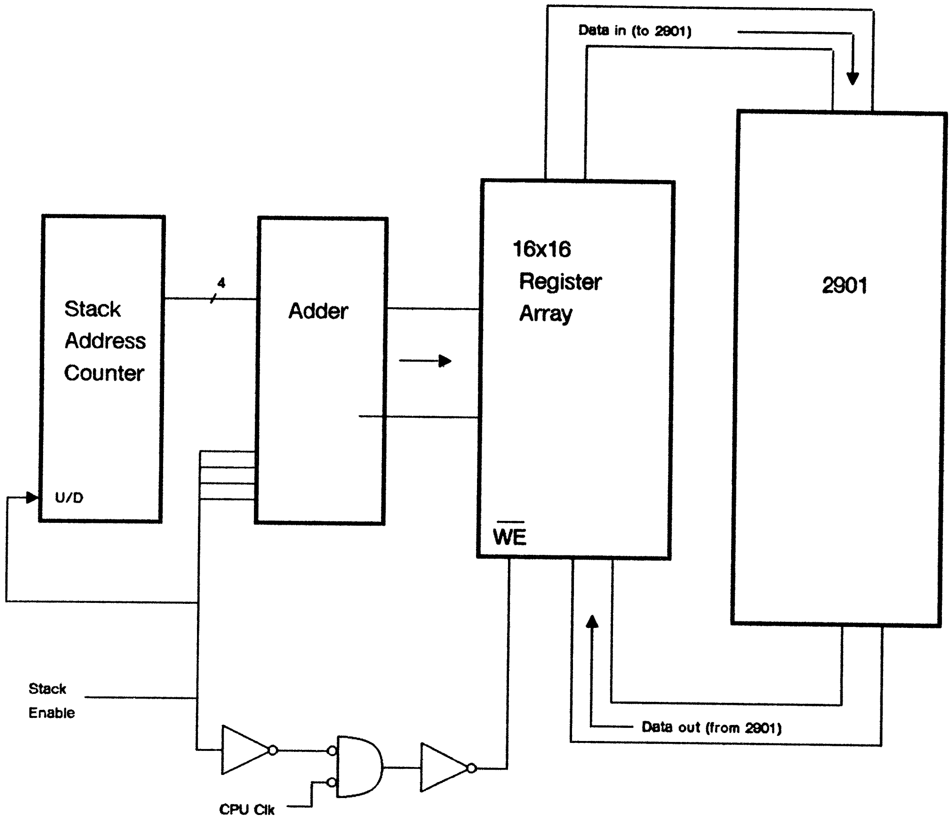


Figure 5.5 Operation of Evaluation Stack

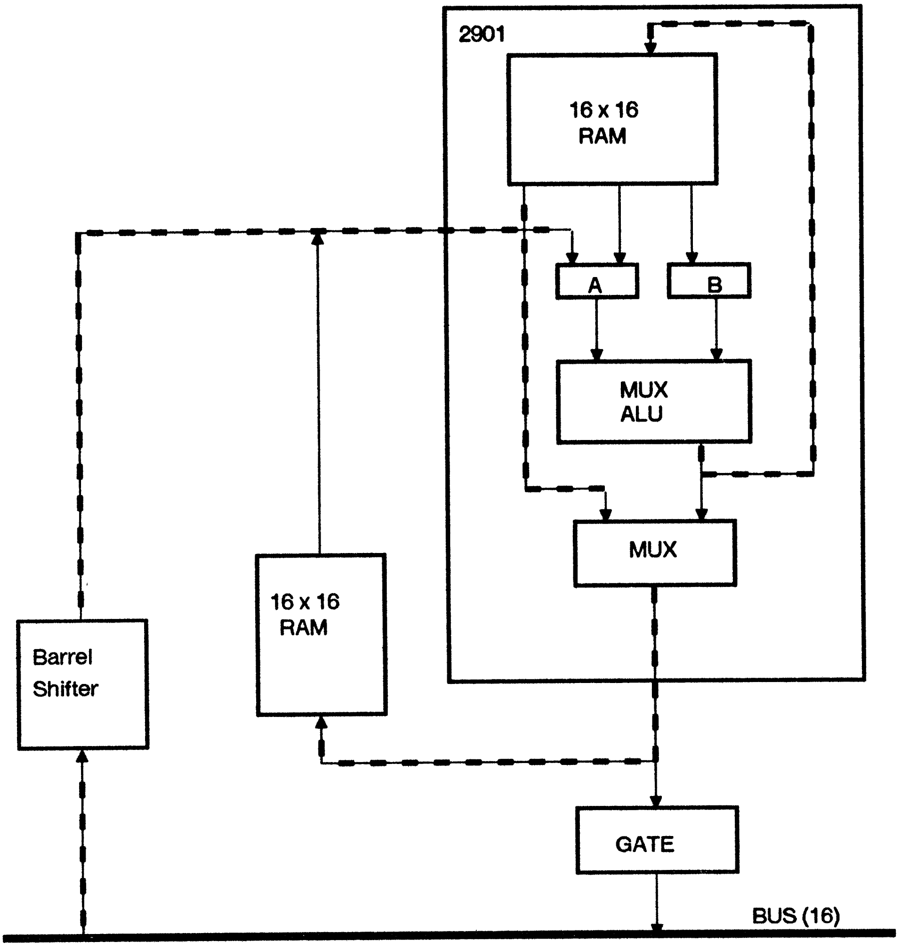


Figure 5.6a Expression Stack Push Operation

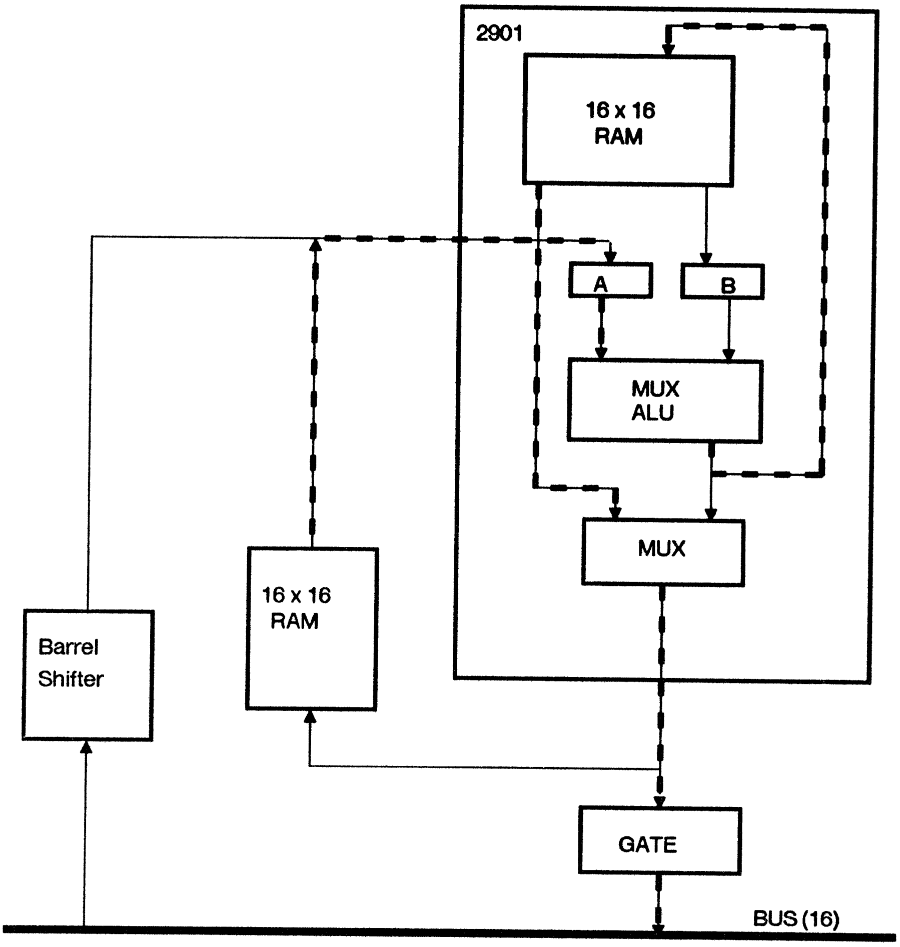


Figure 5.6b Expression Stack Pop Operation

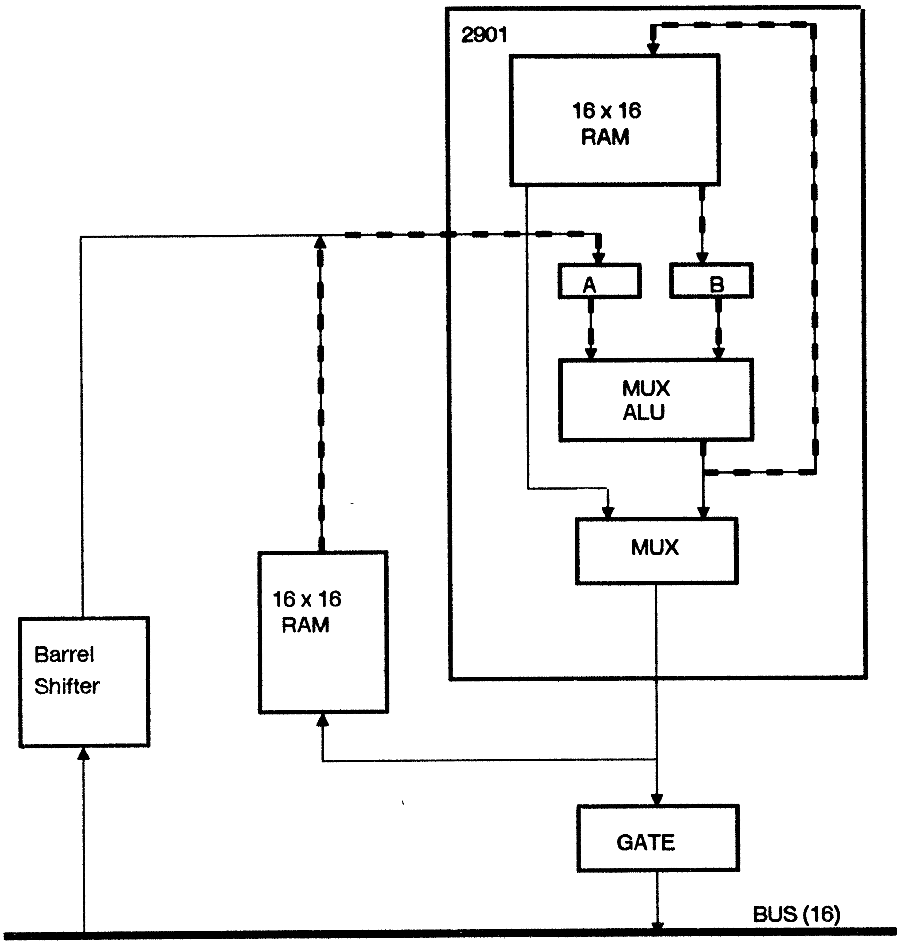


Figure 5.6c Expression Stack Dyadic Operations

configuration which would provide sufficient stack depth which would not also add the overhead of testing for stack overflow. Thus, increasing stack depth could not eliminate the need for saving the stack in operations where the compiler could not verify or control the depth of stack usage. As a consequence, it was concluded that sixteen levels is actually adequate, and that a greater number would most likely be of little value. As it currently stands, the evaluation stack is saved into the main memory stack whenever any operation takes place in which the compiler cannot control the depth of stack usage.

5.4 Design of Memory Organization

It was clear from the beginning that we would need a large bandwidth of memory to satisfy the needs of a high resolution bitmap graphics display. We noted that the memory bandwidth required to operate the bitmap display of the Alto computer was such that whenever the display was turned on the processing power of the machine was reduced to a third of its normal capacity [TMLSB]. This was attributed to the small word width of the Alto memory. To avoid this problem, we arrived at a memory organization design which provides for a 64-bit multi-word read operation, i.e reading four words at a time, and a single word 16-bit write operation. This organization is feasible by using 16k and 64k (k = 1024) memory chips which have separate input and output data lines. While the output data lines are organized to read 64 parallel bits onto a memory bus, the sixteen input lines each go to four different chips in a 64-bit word, the chips having a control signal to select which 16-bit word actually receives the data (Figure 5.7). To provide for 16-bit read operations as well as 16-bit write operations, we eventually added the necessary gating circuitry to select one word of the four and to gate it to the 16-bit memory data bus. This gating circuitry is primarily used by the data port of the processor and the Ethernet interface.

Having the main memory service both a bitmapped screen and a processor created a need for either a multi-port memory or a tight coupling between the display and processor. In the latter alternative, the use of memory would be totally controlled by the processor and given to the display as needed. This concept would also have to extend to any future DMA devices which used the memory of Lilith directly. Instead of this alternative, we chose to provide a multiport memory system having eight possible memory ports. We defined a protocol where the requesting port would assert to the memory allocation circuitry a request for a memory cycle. When the cycle is available, the memory would acknowledge the signal and assert a selection line in return, telling the port that it should place its address, and data if necessary, onto the memory address and data buses. The memory control would then provide timing signals to strobe information into registers in the port and to clear the request for a memory cycle at an appropriate time within the cycle. The memory timing circuitry itself was constructed from a small state machine which also asserted at appropriate times the necessary row and column strobes to the memory chips and strobed the data from the memory into the port register appropriately.

5.5 Design of the Instruction Fetch Unit

At first we had no intention of having an a special unit for instruction fetches. However, our desire to implement an instruction set based upon byte operation codes conflicted with the concept of a word-addressed memory. Some effort was expended in designing schemes whereby byte-sized instructions could be fetched from a double byte wide memory without the overhead of an excessive number of memory cycles. A very simple but inefficient approach to this problem involves reading a word of memory for every byte desired and subsequently selecting that byte while discarding the other byte. This technique would have been far less costly in hardware, but also extremely inefficient by comparison to methods we could envision using a separate unit to fetch instructions. We finally concluded to accept the extra cost and use the extra memory bandwidth to develop the *Instruction Fetch Unit*. It provided us the necessary addressing to each byte of a word and allowed us the possibility to fetch eight bytes of instruction at a time.

At first we considered the possibility of having a single 16-bit counter which would be a byte

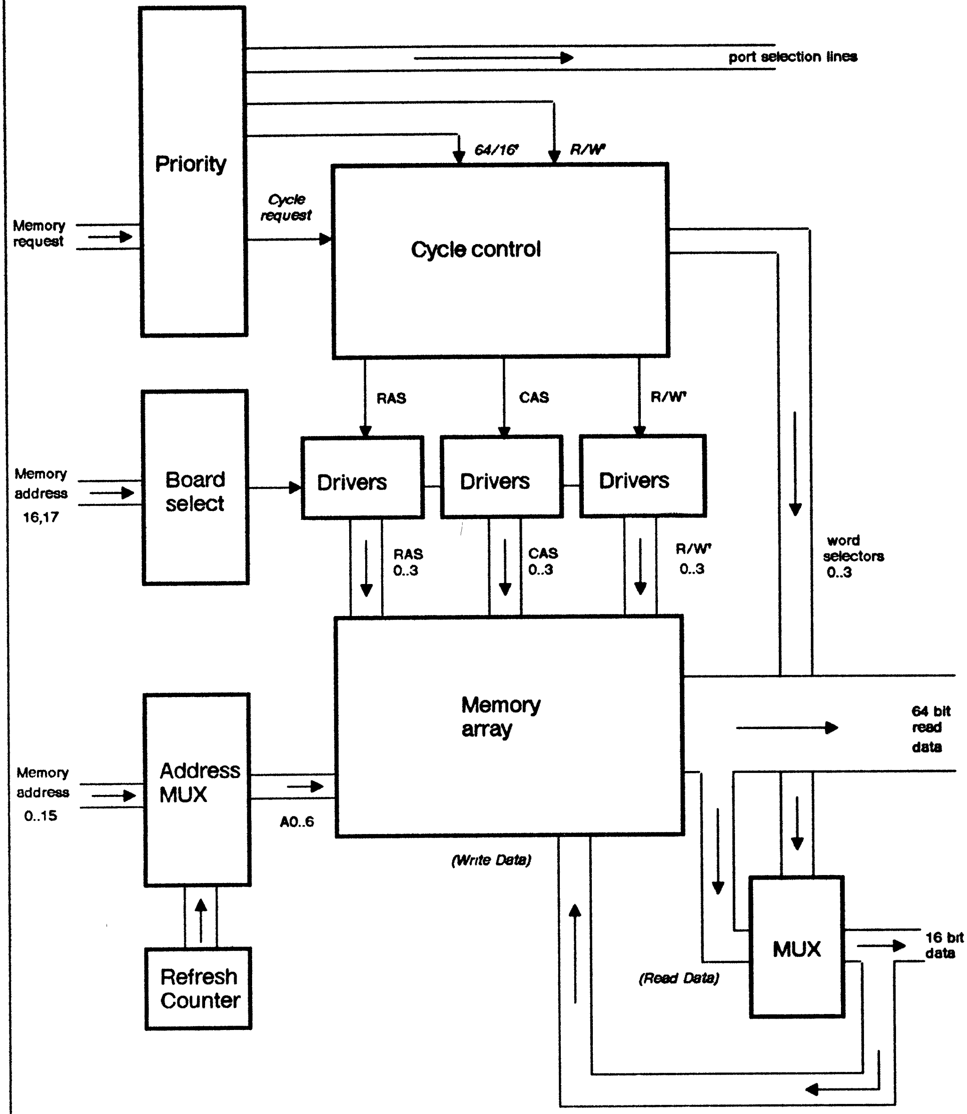


Figure 5.7 The Lilith Memory

address counter. This, of course, would have only allowed access to the first 64k bytes of memory for use as program storage. It did, however, allow manipulation of the program counter as a single word of memory. After some deliberation, we concluded that this was not acceptable and began looking at techniques which would allow us to use a base register value and an offset. In the end, we developed a technique using two registers as a means of computing the final program counter address. One register is the so-called frame register, which provides the base address where any module of code is located. The frame register has a 16-bit value and is therefore storable in a single word of memory. To provide the capability for locating instructions beyond the first 64k addresses of memory, the frame register was considered to be shifted with respect to the actual program counter address. The bits not included in the frame register are presumed to be zeros. This required that each module begin on an even or *multiple of 4* address.

The second register, called the offset register, is also a 16-bit address which, when added to the frame register shifted by two bits, provides an eighteen bit byte address that selects the instruction byte to be used next. The hardware to implement this structure included a complete 20-bit adder circuit. Philosophically, we had a hard time accepting this design. After all, this was supposed to be a simple machine, and it had a complete 20-bit arithmetic unit in the instruction fetch unit!

The IFU required of the microprogrammer a conceptual change in his understanding of instruction fetching. Instead of performing instruction fetches by giving the address of the byte desired and waiting for the instruction to be returned, the microprogram is forced to use the philosophy of saying, "Give me the next instruction," without knowing the actual address of the instruction. Only when a branch operation is to take place does the program have any contact with the instruction address. In this case, it would reload the offset register as well as possibly the frame register. The logic of the IFU is complicated by the fact that the processor does not know whether or not it has the requested instruction byte within the IFU's eight register storage area. As a consequence, the IFU is required to automatically fetch the next group of eight instruction bytes as soon as the instructions from the previous group of eight have all been utilized.

One of the desirable features of this addressing technique is that it allows program instructions to be found in larger areas of memory than the 64k addressing space of a 16-bit machine. It does so in such a fashion that the programmer is unaware that his program resides outside of normal 64k boundaries. This is not the case in many 16-bit minicomputers, which have paging mechanisms. In Lilith, the use of memory pages beyond the initial 64k words for instructions is totally transparent to the programmer. The Instruction Fetch Unit is shown in Figure 5.8.

5.6 Design of the Clocking Circuitry

In the simplest case, the clocking circuitry is nothing more than a free running oscillator with mechanisms to stop it or to *single step* it for debugging purposes. In Lilith, the clocking circuitry has been complicated by the design decisions which have previously been discussed in this chapter. These complications are outlined as follows.

First , the use of a map ROM for decoding M-codes causes the execution of the M-code decode instruction to take a longer time than the normal arithmetic operations. Because of this, we were forced to develop a clock circuitry which had long and short microinstruction times; the long is 225 nanoseconds and the short is 150 nanoseconds. The selection of a microinstruction's speed is controlled by a bit in the microinstruction format selects between long and short instructions.

A second problem was created by the possibility that the multiport memory interface might allocate memory cycles to other devices outside the processor's control. Consequently, the possibility exists that the processor might request a memory cycle at a time when it would expect to find the memory data available but instead would find that its memory request had not yet been granted.

Two possible solutions to this problem were considered. The first was to provide a status bit which the processor could test to determine whether or not the memory data port had completed

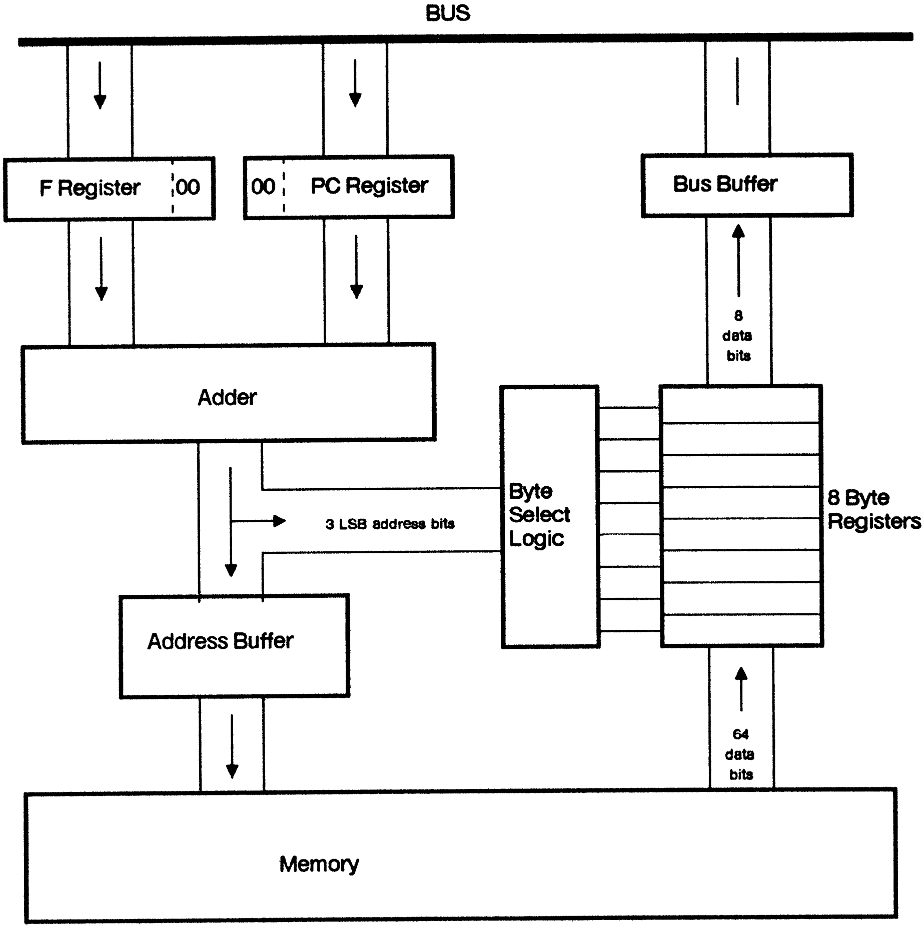


Figure 5.8 Instruction Fetch Unit

its assigned operation. This bit would have to be tested before initiating any operation and would also have to be tested before retrieving the results of any read operation.

The second possibility was to include in the clock circuitry the capability to simply halt the clock for a period of time whenever the processor attempted to execute a microinstruction referencing a memory port which had not yet completed its operation. We chose the second technique rather than the first because we felt that there would be too many microinstructions wasted in testing the status of the memory bit, and we also felt that there would be time lost for unnecessary tests of the memory port, which was already available.

There were, in fact, two memory ports to consider: first, the *data port*, which was used for reading and writing 16-bit words of data into memory, and second, the instruction fetch unit, which provided the processor with bytes of memory for use as M-code. Because of these considerations, the clock circuitry of Lilith became a small state machine with a total of three states. The sequencing of this small state machine was controlled by "ready" signals from the two memory interfaces and commands from the current microinstruction which determined whether or not an "unready" memory interface was being used in that microinstruction. Figure 5.9 depicts this circuitry.

5.7 Design of Interrupt Circuitry

We had been very impressed by the clever usage of the interruptible microinstruction control unit of the Xerox Alto computer [TMLSB]. It had succeeded in minimizing the hardware needed for its I/O devices by off-loading much of the work to a main processor which was able to respond virtually instantly because of its interrupt structure. We considered for some time using this idea, but finally decided it would be extremely difficult to fit the idea of interruptible microinstructions into our design.

The next logical choice, then, seemed to be to allow interrupts to occur only at end of an M-code. It was relatively easy to implement this as it requires only a small amount of circuitry to detect the interrupt requests and to generate a "hard-wired" vector for the next microinstruction address. This "hardwired" vector replaces the next microinstruction address which would have normally come from the map ROM which decodes M-codes. To make things easier for the programmer of the microcode, we also inhibited (during an interrupt response) the command to the IFU which causes the program counter to advance. By inhibiting the *advance* command to the program counter, the correct state of the machine was preserved for the interrupt handler.

To provide the processor with the ability to mask interrupts whenever it desired, an interrupt mask register was included which can inhibit any or all interrupts according to the data loaded into it from the processor bus.

It was an oversight that we did not anticipate how complex some of the M-code instructions would be. Some of them are composed of hundreds and even thousands of microinstructions. This causes problems with some of Lilith's I/O devices. The RS-232c interface for example, sometimes loses characters because of the processor is tied up with an unusually long M-code. Had we anticipated this, we would have designed some of the I/O devices differently.

Figure 5.10. shows details of the construction of the interrupt circuitry.

5.8 Design of Special Short Opcode Masking Instructions

After we began programming the microprogram which was to perform the M-code interpretation, we discovered that many instructions were being used in a potentially superfluous manner to obtain the short constants which were imbedded in some of the 256 M-codes. Knowing that these M-code instructions would be used quite frequently, we began an investigation to see if ther was some way we could optimize these operations. Fortunately, the design, as it had progressed until that date, could still be modified without a major effort. We found that these four bit constants could be stripped out of the M-code during the microinstruction cycle which fetched and decoded the M-code from the IFU. Previous to this time, the M-code had passed directly from the IFU to the MCU. All that was necessary to allow the constant to be captured was

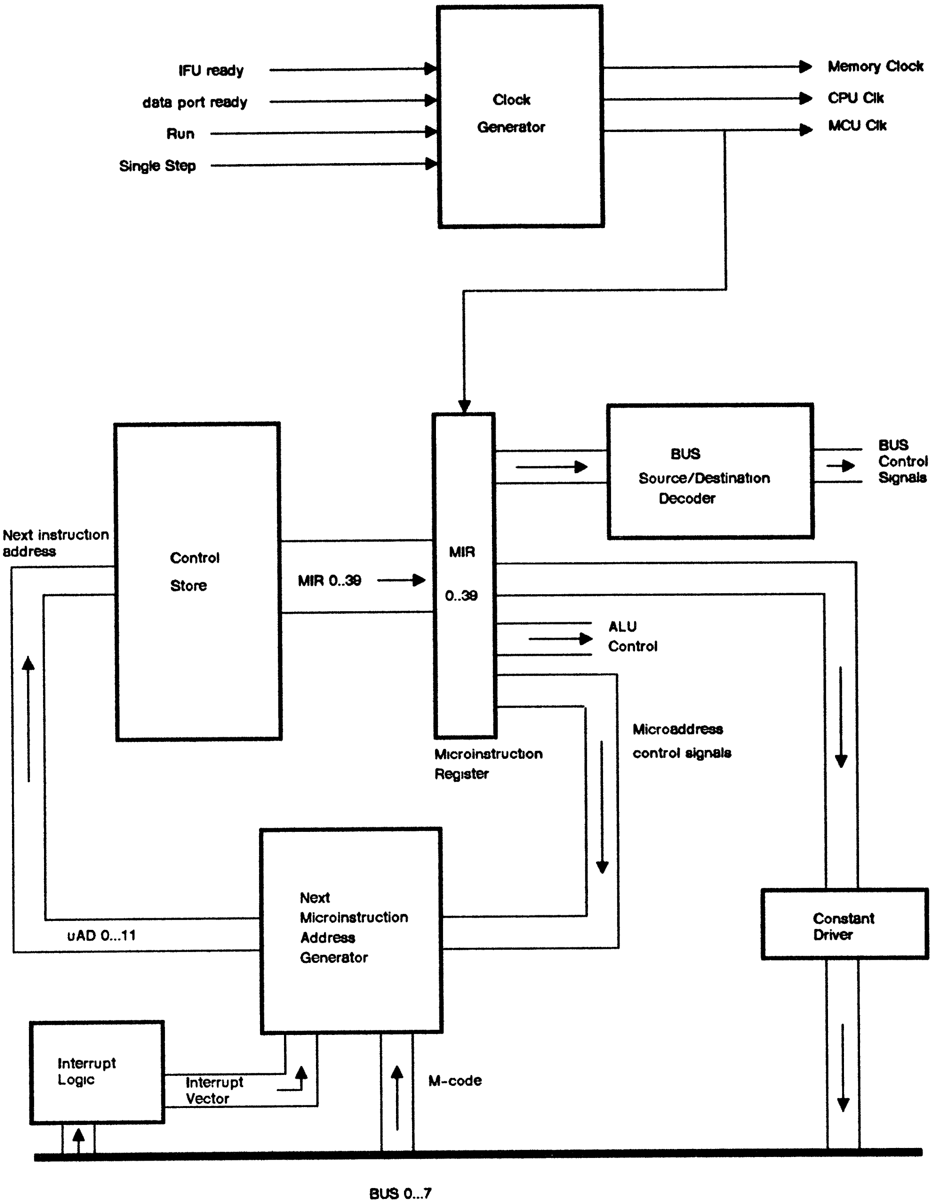


Figure 5.9 Micro Control Unit Block Diagram

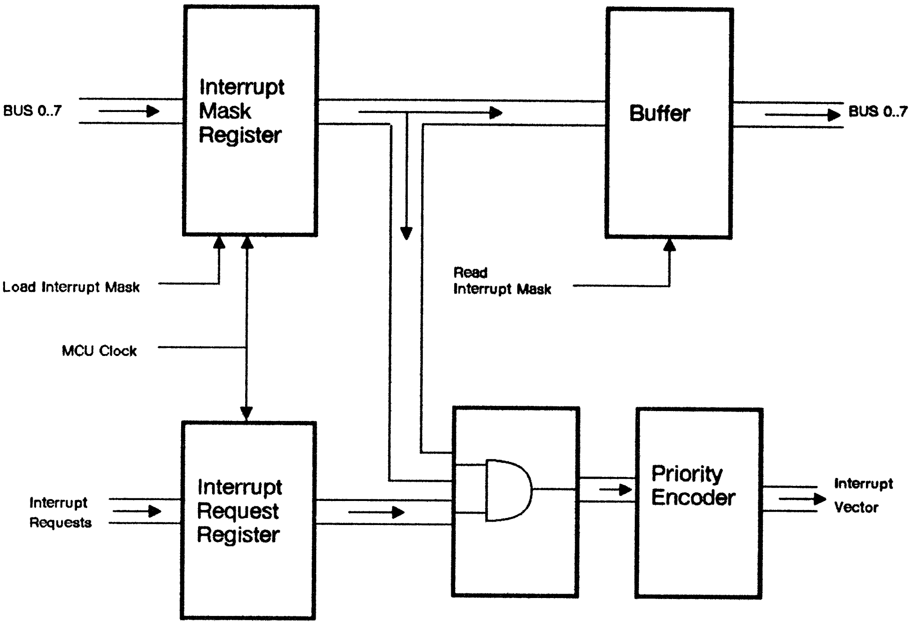


Figure 5.10 Interrupt Reguest Handling

to pass the M-code over the bus and to include proper commands in the M-code decoding microinstruction so that the contents of the bus were loaded into the Q register of the 2901. In the ALU, a modification was made so that the use of the "jump map" instruction caused the data coming from the bus to be stripped of any significant bits above the least significant four.

The effect of this modification, allowed all M-codes using short constants to be targeted to the same entry microinstruction for interpretation. And, no further cycles were necessary to obtain the short constant for use. After the fetch and decode of an M-code, the 4-bit constant field from the M-code would be ready for use in the Q register of the ALU. Of course, if an M-code were not one of those using short constants, then the contents of the Q register would simply be ignored.

5.9 The Design of the High Resolution Bitmap Display

Much effort went into the design of the high resolution bitmap display interface. We expected the display to burden the processor with incessant memory cycle requests to refresh the image on the screen. With this in mind, we included features in the display interface which would allow us to reduce the size of the visible bitmap both vertically and horizontally. Also, a reduced size bitmap could be vertically positioned almost at will on the screen, and horizontally on quad-word boundaries. Initially, a total of 454,656 bits were displayable on the screen. These were organized as an array of 768 bits horizontal by 592 vertical. Later versions of the display had a *portrait* mode organization with 704 dots horizontal by 928 dots vertical.

We did not include a hardware character generator in the display interface. Therefore, for all characters which were to be displayed upon the screen, it was necessary to paint them from images stored in memory. The character images were kept in special data structures which were appropriately called "fonts". A specially designed microinstruction "understands" the structure of font files in memory and can transfer character images from the font files to the display with great efficiency.

Our apprehensions about the burden placed upon the memory by the display interface proved to be unfounded. The following chapter will be deal with this topic more analytically. It suffices here to say that the features of the display interface which allow us to reduce the image have been seldom used, and in a later redesign of the interface, these features have been completely eliminated.

5.10 Diagnostic Processor Unit (DPU) Design

As we began to diagnose the first wirewrap version of the processor, we quickly concluded to employ a simple microcomputer to help us trace the execution of instructions. The idea worked poorly at first because of our "jury-rigged" implementation. However, the potential value of such a diagnostic processor did not escape us. Very quickly we abandoned the "jury-rigged" implementation and concentrated our efforts on an improved version using a Motorola 6802 processor. This resulted in a *diagnostic processor unit* (DPU) which has the following capabilities:

1) It can disable the microinstruction register (MIR) of the MCU and substitute its own microinstruction.

2) It can disable the address to the microcontrol store ROM and substitute its own.

3) It can inhibit the clock to the ALU regardless of what type of microinstruction is being executed.

4) It can allow a single or a controlled number of clock pulses to occur .

5) It can be selected as the bus source.

8) It can read the MIR, next microinstruction address, or the bus.

With the appropriate software, the DPU can perform the following high-level functions:

1. It can load test programs in memory for Lilith execution.

2. It can execute single microinstructions in a controlled fashion and verify their correct operation.

3. It can allow single step execution of microinstructions and M-codes for debugging microprograms.

4. It can repeatedly execute short sequences of microinstructions at full speed so that the maximum operation speed can be measured

In normal usage, the DPU board is not installed. When installed, the machine can only be booted by its command. Normally, the DPU would only be used by maintenance engineers and microprogram writers. See Figure 5.11.

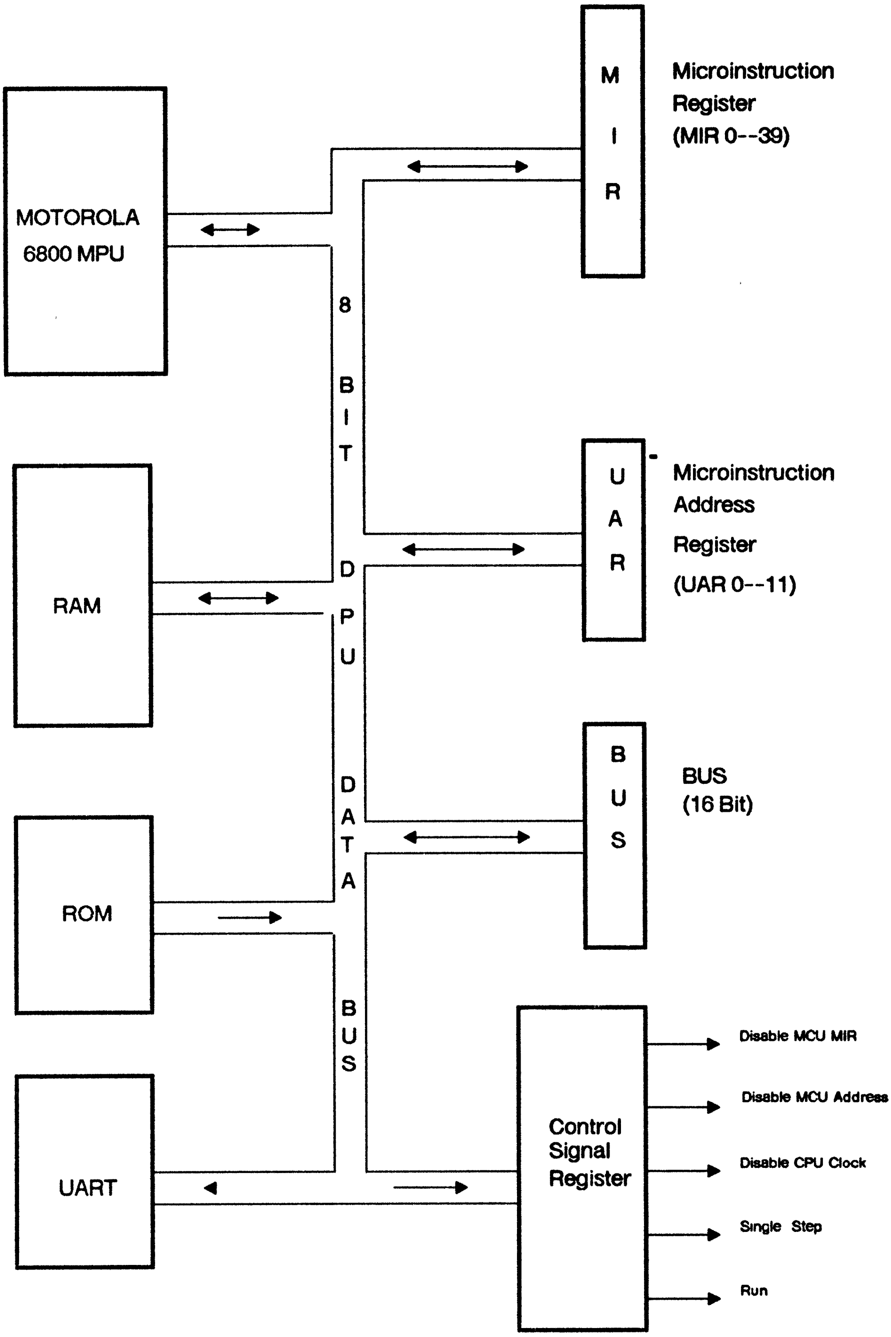


Figure 5.11 DPU Block Diagram

6 Evaluation of the Lilith Design

The design decisions of the Lilith chronicled in Chapter 5 were made on the basis of information and materials available at the time. This chapter will evaluate the Lilith design in the light of current knowledge and three years experience working with Lilith.

6.1 Simulation Program

In order to get detailed measurements of the internal workings of Lilith, a simulation program was written which captured quantitative information about the execution of a Lilith program. A description and listing of this program is given in Appendix 2. Use of this program yielded quantitative information about the operation of Lilith such as the number of microinstructions and memory cycles involved in the execution of a program. The results reported in this chapter come from the simulated execution of a widely known and used benchmark program [Wir2] [Jac]. The benchmark program is listed in Appendix 3.

The simulated execution of the benchmark program lasted only 36.09 milliseconds according to the simulation calculations. In actual execution, one thousand iterations of this program were executed so that the actual execution rate could be accurately timed. The actual timed rate of execution for one thousand iterations was an average of 36.48 seconds. The discrepancy is only about one percent, which is attributable to memory refresh interference and the error in measurement. Also, the execution time of the FOR loop which caused the program to execute one thousand times was not calculated in the simulation. Finally, the execution times of Lilith can vary significantly because of variance in the code position relative to the *quad-word* boundaries of the memory. In view of these uncontrollable factors, greater accuracy from the simulation could not be expected. The results of the measurements in the simulation are given in Table 6.1.

item	description	count	units
1	Total Execution Time	36.09	milliseconds
2	Processor Instruction Time	25.89	milliseconds
3	Processor IFU Wait Time	1.83	milliseconds
4	Processor MDP Wait Time	8.38	milliseconds
5	Microinstructions Executed	159,549	instructions
6	Test-Format Microinstructions	20,618	instructions
7	Constant-Format Microinstructions	2,604	instructions
8	IFU Bytes Fetched	32,147	bytes
9	M-code Opcode Bytes	26,098	op-codes
10	Short Constant M-codes	19,495	op-codes
11	IFU Memory Cycles	6,396	cycles
12	Memory Read Cycles	21,950	cycles
13	Memory Write Cycles	10,027	cycles
14	ALU Push Operations	13,594	operations
15	ALU Pop Operations	3,460	operations
16	ALU Dyadic Operations	10,134	operations
17	IFU Idle Time Before M-code Decode	3.31	milliseconds
18	IFU bytes Delayed by Memory Access	5,020	bytes
19	M-codes with 2 microinstructions	4,657	M-codes
20	" " 3 "	12,546	M-codes
21	" " 4 "	2,764	M-codes
22	" " 5 "	1,205	M-codes
23	" " 6 "	1,669	M-codes
24	" " 7 "	1,150	M-codes
25	" " 8 "	412	M-codes
26	" " 9 "	0	M-codes
27	" " 10 "	0	M-codes

28	"	"	11	"	1	M-codes
29	"	"	12	"	0	M-codes
30	"	"	13	"	1	M-codes
31	"	"	14	"	413	M-codes
32	"	"	15	"	0	M-codes
33	"	"	16	"	0	M-codes
34	"	"	17	"	0	M-codes
35	"	"	18	"	5	M-codes
36	"	"	19	"	5	M-codes
37	"	"	20	"	1,276	M-codes

Table 6.1 Quantitative Measures of Various Parameters
Taken from a Simulated Lilith Program Execution

A further evaluation of Lilith's performance was obtained in the execution of the individual elements of the benchmark program given in Appendix 2. This same experiment is described in the paper by Wirth and the dissertation by Jacobi. The version of Lilith in this case was improved over both previous measurements, having a faster memory with a cycle time of 375 ns. The results of the new Lilith performance together with the old performance figures are listed in Table 6.2.

Test	Original Lilith	600 nsec Lilith	375 nsec Lilith	VAX 750 Pascal*
a empty REPEAT loop	321	422	464	463
b empty WHILE loop	334	419	480	497
c empty FOR loop	422	528	652	363
d CARDINAL arithmetic	187	218	226	160
e REAL arithmetic	130	144	147	84
f sin,exp,ln,sqrt	87	99	103	51
g array access	109	136	161	55
h with bound test	89	111	126	55
i matrix access	197	249	269	112
j with bound test	164	198	214	112
k call of empty procedure	144	187	214	64
l with 4 parameters	94	123	144	56
m copying arrays	63	96	113	85
n access via pointers	125	162	181	215
o reading a disk stream	80	94	117	

* VAX 750 measurements made by Roger Vossler, TRW, Redondo Beach, CA.

Table 6.2 Lilith Performance Measurements (test repetitions per minute)

As can be seen from the figures in Table 6.2, the latest version of Lilith significantly outperforms the earlier versions, and clearly outperforms the VAX 750 as measured by this benchmark program.

With the information from Table 6.1 and Table 6.2, we can attempt to answer some interesting questions about Lilith such as:

1. How much execution time does the bipolar stack adaptation of the 2901 bit slice processor save?
2. How much execution time does the short-constant masking circuitry of the ALU save?
3. How many test-format microinstructions are used?
4. How often are the constant generation format microinstructions used?
5. How many microinstructions per M-code on the average?

6. What is the average execution rate of M-codes?
7. What are the lengths of the most frequently executed M-codes?
8. What is the average length of an M-code in bytes?
9. How many M-codes are executed per IFU memory cycle?
10. How many M-code fetches from the IFU delay the processor execution?
11. How much performance increase would come if the M-code decode were pipelined?
12. How many memory read cycles are saved by the IFU?
13. What is ratio of instruction fetch memory cycles to data movement memory cycles?
14. How much execution time is due to memory delay?
15. How much of the memory bandwidth is used by the processor?
16. How much interference is there between the memory usage for display refresh and the memory usage by the processor?

The answers to these questions will help draw conclusions regarding the soundness of the Lilith design and help formulate recommendations for the next design iteration.

6.2 Expression Evaluation Stack

The special bipolar expression evaluation stack which has been adapted to the AMD 2901 bit-slice arithmetic unit functions very efficiently because of a feature included in the 2901 integrated circuit. This feature, a path from the register file of the 2901 to the outputs, allows all stack operations to take place in a single microinstruction. This includes push, pop, and dyadic operations. Strangely, this valuable feature appears to have been omitted from later AMD bit-slice circuit designs, i.e. AMD2903 and AMD29116. It appears possible to incorporate a bipolar stack in designs using these processors, but impossible to perform all stack operations in a single microinstruction. First of all, we should like to know how valuable the bipolar expression stack is in comparison to a memory implemented stack, and secondly, whether or not the AMD 2901 implementation is of significant value.

Entries 14,15, and 16 of Table 6.1 gives the number of stack operations for the simulated benchmark program execution. If the expression evaluation stack were implemented in main memory, there would be at least one memory reference for each stack operation and several microinstructions for maintenance of stack pointers. As a minimum, we could assume additional execution time for a single memory cycle and a single microinstruction. Calculating the additional execution time, we arrive at the following:

27188 stack operations * 525 nanoseconds = 14.27 milliseconds

Comparing this figure to the overall execution time of 36.1 milliseconds, we compute the performance decrease due to the lack of the bipolar stack:

14.27 / 36.1 = 39.5 % performance decrease

If we presume the use of a bipolar expression evaluation stack with another (other than 2901) arithmetic logic chip, we can calculate in the same manner the execution penalty caused by the need for an additional microinstruction for stack operations:

27188 stack operations * 150 nanoseconds = 4.07 milliseconds

4.07 / 36.1 = 11.2 % performance decrease

This performance penalty can be further decreased if it is possible to hide the execution of a second microinstruction for stack push operations in the "shadow" of the memory operation necessary for loading variables. Unfortunately, this will not work for loading constants onto the stack, since most of the time they come directly from the IFU without any delay. Consequently, we may presume that the final execution time penalty will be somewhere between 6 and 11 percent.

6.3 Short Constant Masking Circuitry

Special circuitry of the ALU strips 4 bits from every M-code as it is decoded by the MCU. These 4 bits are placed in the Q register in case they are needed by the opcode which is being decoded. Not all opcodes need these constants, but for the simulated execution of the benchmark program, 19,495 M-codes (item 10, Table 6.1) out of 26098 executed M-codes (item 9, Table 6.1) were of the type which used the short constants imbedded in the opcode. This is 74.7 percent. Since this operation takes place simultaneously with the decode of the M-code, there is no further execution time needed before use. If, however, this circuitry had not been included in Lilith's design, it would have been necessary to generate these constants with the constant-format microinstructions. Then, it would have required one additional microinstruction to branch to the microprogram segment which handled the class of instructions having the short constant. This would mean an additional 300 nanoseconds execution time penalty for each short-constant type M-code. The performance decrease for our simulated program is computed as follows:

19495 short-constant M-codes * 300 nanoseconds = 5.85 milliseconds

5.85 / 36.1 = 16.2 % performance decrease

The lack of this circuitry would also impose an additional penalty in the number of microinstructions in the microprogram. Since the use of the circuitry allows the map ROM to target all short-constant M-codes of a similar type to a single control store address, far fewer steps in the microprogram are required. Without this circuitry, two additional microcodes would be required for all short-constant M-codes except for the first one of each class. This would require an additional 212 microinstructions, more than 10 percent of the currently used number of microinstructions in the microprogram of Lilith.

6.4 Microinstruction Format

The condition codes which result from the execution of any microinstruction cannot be tested until the following microinstruction because they are first loaded into a register. The purpose of this register is to enable Lilith to operate at a higher clock rate by limiting the signal propagations which must take place within a single cycle. Without this register, the microinstruction cycle time would be constrained by 1) the signal propagation delays necessary to execute an arithmetic operation and generate condition codes, 2) the propagation delays necessary to use the condition codes to select the next microinstruction address, and 3) the propagation delays necessary to access the next microinstruction from the control store ROMs. With the pipeline register, the signal propagation chain is broken in two with the arithmetic propagation delays working independently from the next microinstruction access delays. Figure 5.2 illustrates this situation.

As indicated in chapter 5, the design decision to use a pipeline register was based upon the premise that the number of instructions requiring condition code tests would be significantly fewer than those not requiring condition code tests. Therefore, those not requiring condition code tests would not be penalized with an execution time long enough to accommodate the test. The savings in execution time of these microinstructions was estimated to provide a net gain in speed in spite of the loss in speed caused by the use of two microinstructions to handle operations requiring the test of condition codes.

In item 6, Table 6.1, the statistics of the simulated benchmark program show that only 20,618 *test format* microinstructions were executed out of a total 159,549 microinstructions (item 5, Table 6.1). Consequently, there are approximately 7 normal microinstructions executed for each *test format* microinstruction. This means that if the microinstruction execution time were extended to allow condition code testing within the same microinstruction as they are generated, then only seven of these microinstructions would be required to accomplish the same function that the 8 pipelined microinstructions are now accomplishing. In order to have the same total execution time, this extended microinstruction cycle could therefore be only 1/7 longer than the current microinstruction cycle time, i.e. 171 nanoseconds. Measurements have shown that this could not be the case since the combined signal propagation delays would extend well beyond 200 nanoseconds. Therefore, the final conclusion is that pipelining the condition codes, as it now exists, is effective for the architecture of Lilith.

6.5 Constant Generation Format Microinstructions

It is significant that only 2,604 microinstructions (item 7, Table 6.1) out of a execution total of 159,595 microinstructions (item 5, Table 6.1) were of the constant generation format. Since this is only 1.6% of the total execution of the program, it clearly represents an opportunity to reassess the impact that the logic implementation of the constant-format microinstruction has had on the overall speed of the machine. Any mechanism that could improve the overall performance of the processor by as little as 5 per cent could well be worth the change, for the time expense incurred by the process of generating constants would be minimal in view of their infrequent use.

6.6 M-codes

The efficiency of M-codes has been covered in other publications [Wir2]. Jacobi [Jac] gives some interesting comparisons in appendix 5 which show comparisons of Lilith code lengths and executions speeds with other well known processors. Therefore, it has been well documented that the compactness of Lilith M-codes gives it an advantage over other processors. What has not been documented is the profile of M-code interpretation by microinstructions.

Using the results of the simulated benchmark program execution, the number of microinstructions executed per M-code were tabulated and are presented in Figure 6.1. An unanticipated result of this tabulation is the fact that 66% of the M-code instructions complete execution in 3 microinstructions or less. The average number of microinstructions per M-code execution is about six, but that is because the number of M-codes with 20 or more microinstructions is significant. Floating point instructions are found in this category.

Dividing the total execution time, 36.1 milliseconds, by the total number of microinstructions executed, 159,549, gives the average execution time of a microinstruction, which is 226 nanoseconds. This gives an average execution time of 1.38 microseconds per M-code, or an overall M-code execution rate of 725,000 instructions per second. This figure appears artificially low because of the high number of microinstructions executed in some of Lilith's more complex M-codes. A more realistic figure for comparison would take the average of the main body of M-codes which is approximately 3.5 microinstructions per M-code. This gives a figure for comparison of 1,260,000 M-code executions per second.

6.7 The Instruction Fetch Unit

It was hoped from the beginning that the IFU would be able to deliver instructions to the processor with greater efficiency because of its use of the 64-bit memory interface. Since it fetches 8 instructions at a time, it was expected that memory references required to provide instruction bytes would be significantly lower than the number of bytes delivered. For the simulated execution of the benchmark program it was determined that 6,396 memory cycles (item 11, Table 6.1) were necessary to deliver 32,147 instruction bytes (item 8, Table 6.1) to the processor. This is an average of five instruction bytes delivered for each memory reference. For the most part these bytes were delivered without causing a delay in the processor execution. In fact, the total delay in processor execution (out of the total of 36.1 milliseconds of execution time) was only 1.83 milliseconds (item 3, Table 6.1), or 5.06 percent.

Close inspection of the IFU delay figure, 1.83 milliseconds, would appear low considering 6,396 memory cycles (item 11, Table 6.1) each requiring 375 nanoseconds for a total of 2.39 milliseconds. To reconcile this, one must recognize that each time the IFU delivers the last of a block of 8 instruction bytes, it automatically prefetches the next block. Consequently, it is entirely possible that when the processor arrives at a point where it needs another instruction, that instruction has already been fetched and may be delivered without any delay. Using the simulation program, the count of IFU instruction requests delayed by memory access times was

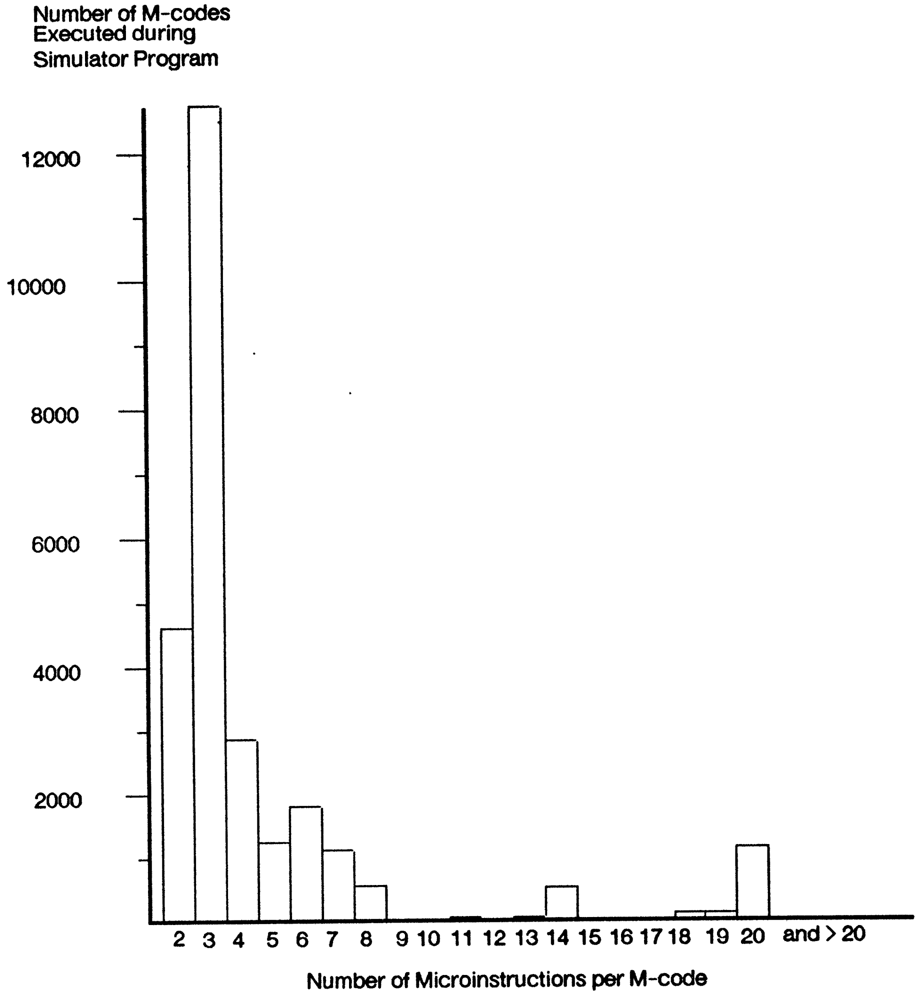


Figure 6.1 Executed M-code Distribution in Simulator Program

determined to be only 5,020 (item 18, Table 6.1) for the benchmark simulation.

If one were to view the IFU as a simple cache memory with the criteria that an undelayed instruction delivery constitutes a "hit," then the IFU would appear to have a "hit" ratio of 84.3 %.

(32,147 ifu bytes - 5,020 delayed ifu bytes)/ 32,147 ifu bytes = 84.3%

It is also interesting to note that of the 32,147 ifu bytes (item 8, Table 6.1) delivered, 26,098 (item 9, Table 6.1) of them were opcodes, giving an average of 1.23 bytes per M-code instruction.

In summarizing the effectiveness of the IFU, one should note that if the IFU could accomplish its task perfectly (no delays for any instruction fetches), the actually processor performance would only be increased by 5.06%. This indicates that there are better places to begin optimizing Lilith than in the IFU.

6.8 M-code Decode Delay

Because of the additional signal propagation required to allow the M-code map ROM to deliver the starting address for the portion of the microprogram to interpret an M-code, the Lilith processor allows an extra long microinstruction cycle of 225 nanoseconds. Because of the pre-fetch capability of the Instruction Fetch Unit, it is conceivable that the decode of M-codes could be overlapped with the execution of the previous M-code. Therefore, it would be of value to evaluate the percentage of execution time occupied by M-code evaluation. Using the tally 26,098 M-codes for the simulated executions, we find that 5.87 milliseconds were spent decoding M-codes:

26,098 M-codes * 225 nanoseconds per M-code decode = 5.87 milliseconds

which is 16.27 % of the actual execution time. We would find this performance improvement if the M-code decode delay could be completely overlapped with the previous instruction execution. Of course, this may not be possible if the IFU were incapacitated by an ongoing memory cycle. In order to determine the potential possibility for overlapping instruction decode, the simulation program was modified to include the counting of IFU idle time prior to M-code decode operations. The resultant calculations show a potential for a 14.6 reduction of execution time if the decode operation execution time could be reduced to 150 nanoseconds and overlapped as much as the memory would allow with the execution of previous M-codes.

6.9 Memory Efficiency

Von Neumann introduced the concept of storing program instructions as well as data in memory. Unfortunately this leaves the processor continually waiting for memory access. For this reason, memory usage in computers has often been referred to as the "Von Neumann bottleneck. Using the simulation program, we will now analyze Lilith's pattern of memory usage to see if it suffers from this syndrome as much as conventional processors.

First of all, it is of interest to calculate the overall performance degradation caused by memory access delays. From item 2 of Table 6.1 we see the total computed execution time for the program according to the theoretical time allotted each microinstruction to be 25.89 milliseconds. The calculated memory access delay for the IFU is 1.83 milliseconds (item 3, Table 6.1) and for the memory data port, it is 8.38 milliseconds (item 4, Table 6.1). This computes to a total of 28.3% of the execution time which can be attributed to the delay caused by memory access.

This is a low figure when compared to conventional processors. It is low because of the Lilith's exploitation of the 64-bit width of its memory for instruction fetch operations. As an indication of the effectiveness of Lilith's memory, consider that a 50% decrease in Lilith's memory cycle time would only yield a 14% increase in processor speed. Clearly, Lilith's performance is more dependent on it's basic processor speed than upon it's memory cycle time.

6.10 Unused Memory Capacity

Having discovered Lilith's usage of the memory to be quite efficient, the next question to address is: What additional memory bandwidth is available for other purposes such as I/O devices and co-processors? The percentage of memory utilization by Lilith can be calculated using figures from the simulated execution of the benchmark program.

First of all, the total number of memory cycles available during the simulated execution period is 96,267.

36,100,000 nanoseconds / 375 nanoseconds per cycle = 96,267 cycles

According to the statistics in item 12 of Table 6.1, 21,960 memory read cycles were initiated as well as 10,027 memory write cycles (item 13, Table 6.1). This gives a total of 31,987 memory cycles used by Lilith from of the available 96,267. The percentage utilization is 33.2%. The refresh operation for the dynamic memory chips requires 128 cycles every two milliseconds, or another 2310 cycles. This occupies another 2.4% of the Lilith's memory bandwidth. So, a total 61,970 memory cycles were available during the period of the benchmark program's execution for other uses, i.e. a total of 64.3% of the Lilith's total memory bandwidth.

6.11 Display Refresh Interference

All measurements of Lilith's performance by simulation or actual program execution as reported in this document have been without usage of the display or consideration of the delay which it would cause with competing requests for memory cycles. Two versions of a high resolution display are currently in use in Lilith. The original version is compatible with the European black and white television standard. The refresh of this display requires 454,656 bits each 40 milliseconds which translates to 7,128 read memory cycles during the period. The percent of Lilith's memory bandwidth used for this operation is computed as:

7,128 memory cycles * 375 nanoseconds / 40,000,000 nanoseconds = 6.68%

The second version of the Lilith display has a screen dimension of 640 by 832 pixels for a total of 532,480 bits. The refresh rate for a complete image is 36 frames per second. A similar calculation for the use of the Lilith's memory bandwidth is:

8,320 memory cycles * 375 nanoseconds / 27,777,777 nanoseconds = 11.23%

If Lilith were using its full memory bandwidth, then one could expect a performance degradation approximately equal to the above percentages. However, since the actual memory usage by the Lilith processor is approximately 33% one would expect only one third of the display refresh cycles to actual interfere with the processor performance. For the execution of the benchmark program which was actually timed at 36.48 seconds without the display, a similar measurement was conducted with the higher speed. portrait display active giving a time of 37.47 seconds. This represents a performance degradation of 2.73% as actually measured. The measured figure is actually lower by .3 seconds than the calculations would predict in this experiment. This is easy to understand but difficult to explain. In this document we shall have to be satisfied to merely state that it is caused by the difference in the synchronous timing relationship of the processor with the memory and the asychronous timing relationship of the display with the memory.

6.12 Processor Efficiency

A measure of a processor's speed is relatively easy to benchmark. A measure of its efficiency is not so easy. Lilith seems to perform 3 to 5 times faster that other processors with similar memory cycle and microinstruction execution times. How to capture and measure this efficiency seems to be a difficult problem. The author of this dissertation wishes to propose a figure of merit which he thinks should reflect the relative efficiency of a processor. The figure of merit proposed is a simple ratio of the number of memory cycles used for data movement and the number of memory cycles used for instruction fetch. The rationale behind this lies in the assumption that most memory operations for data movement are in some way productive whereas memory operations for instruction fetches may be viewed as overhead. Consequently, a greater ratio implies a the greater efficiency. It is predicted that most processors will have a ratio less than one whereas for Lilith, the following ratio is computed from the simulated benchmark program:

31,987 total memory cycles - 6,396 IFU cycles = 25,591 data cycles

25,591 data memory cycles / 6,396 IFU memory cycles = 4.00

If, as expected, other processors do have values less than one for the ratio of the number of memory cycles used in data operations and the number of memory cycles used for instruction fetch, then this figure of merit does explain part of how Lilith attains a greater efficiency than conventional computers.

7 Conclusions and Recommendations

The Lilith design has been presented and evaluated with the help of a simulation program. Now, conclusions will be drawn in this chapter, and recommendations will be made to guide future efforts.

7.1 Conclusions

Stack Architecture is Still a Better Way

More than 20 years have passed since the advent of the first successful stack architecture machine [Bar] [Ham], yet the world at large still seems to question whether or not stack machines are in fact more efficient than the traditional register architecture machines which have enjoyed the greater popularity.

Lilith now adds weight to the accumulated evidence which supports the hypothesis that stack machines are more efficient by demonstrating greater performance than other machines of comparable technology. However, because of the diversity of computer applications and the large number of programming languages in use, this issue may never be resolved. David Bulman has summarized the situation very clearly in a 1977 paper [Bul]:

> For anyone who has been working with stack computers for very long, it is hard to see why any non-stack machines are built at all, except for requirements of machine language compatibility. With the almost total move to high-level languages which economic realities will force on the industry, any such residual interest in machine language should vanish in the near future. Of course, the designers and builders of the B5000 were saying that in 1960.

Instruction Fetch Unit Breaks the Von Neumann Bottleneck

Having an instruction fetch unit with a 64 bit read path to memory in Lilith is certainly one of the most significant contributions to its better performance. Most computers divide their time alternating between instruction fetch operations and instruction execute operations. Hence they spend approximately 50% of their time at each function. The Lilith instruction fetch operations occupy the processor only about 23.8% (see the Figure of Merit conclusion below) of the time. The rest of the time is spent in execution. Without the Instruction Fetch Unit, it would be about 50%. The Instruction Fetch Unit provides one other advantage: it's built-in adder circuit provides automatic base register addressing and consequently allows the code to be entirely position independent in memory. Since the IFU only requires 32 small scale and medium scale integrated circuits (less than 10 percent of the total machine parts bill), it is well worth the investment.

The Expression Evaluation Stack Works Well With the AMD2901

Other stack processors have also had register stacks in the arithmetic logic units. The original Burroughs B5000 had two levels in register with an automatic overflow into main memory. Later versions of the Burroughs computers as well as the HP3000 have had more stack levels implemented in registers with automatic overflow [Bur][HP], but Lilith manages to implement its stack without automatic spill-over. It does this by making the assumption that all programming will be handled by a compiler which will manage the level of stack usage. This also means that no special tests are ever required to determine if there is a stack overflow in the expression evaluation stack.

The AMD2901 implementation of the expression stack is especially efficient since all stack operations can be performed in only a single microinstruction. Since no test operations are ever required to determine stack overflow, most stack operations, including dyadic operations such as add, subtract (but not multiply or divide) can be completed in one or two microinstructions beyond the M-code fetch and decode operation.

The final conclusion is that the implementation of the expression evaluation stack in Lilith is both cost effective and efficient.

Multiple Word Sizes Are Better

Many computer designers allow themselves to be constrained to a single word size for all functions when, in truth, different functions of a computer are better implemented with different word sizes. An instruction size of eight bits was found to be more efficient in Lilith for high-level language use, and a 16-bit word length was selected for data. For I/O memory read operations a word length of 64 was selected to provide a higher data rate and thereby lessen the frequency of memory requests. Because of this, Lilith has only minimal performance degradation due to display refreshes. It is difficult to attribute all the efficiency of the Lilith processor to any one design decision, but certainly the choice of multiple word sizes according to the processor needs was a key contributor.

16 Bits Is Enough

No matter how big the computer is, there will always be a program too big for it to handle. One must learn to think only in terms of the subset of computer programs that are reasonable to be used on a given computer. A 16-bit Lilith computer handles a very reasonable subset of problems--more than enough to make it valuable. That subset, however, probably does not include some of the programs which are currently fashionable. It does include program development, word processing, many graphic programs, and even some computer aided engineering. However, it is questionable whether one has sufficient capacity in Lilith to develop programs for the design of VLSI integrated circuits..

It is a mistake to compare the 16-bit addressing capability of Lilith to the 16-bit addressing capability of some of the more popular microcomputers, and automatically assume that Lilith suffers the same restrictions. First of all, Lilith addresses words, not bytes, and it also operates with base address registers for instruction addressing so that programs may reside without penalty in a much larger address space than 16 bits allows. Furthermore, a number of important data operations such as display addressing, font addressing, and I/O transfer buffer addressing do not need to fall within the 16-bit address limitation. Only addressing of program-declared variables needs to fall within this range, so that they may be passed as parameters by reference in procedure calls.

Some users have proposed that Lilith should be given the capacity to handle an expanded addressing space by allocating two words for a memory resident address as has been done in the Alto and the PERQ computers. By doing this, Lilith would have the capacity to address very large data structures. In fact, Lilith already has the ability to handle double word pointers in a fashion which works very well for large data structures. The problem is that the use of these double word pointers is not directly supported by the compiler, and therefore, must be programmed through the use of the compiler *code procedure* mechanism. This could (and has!) worked very well when data structures have been defined and accessed through special procedures.

Before any serious consideration is given to the idea of attempting to handle 32 bit addresses in the current 16 bit Lilith, one should evaluate carefully the performance difference between PERQ, which does exactly that, and Lilith. Jacobi [Jac] has reported measurements, which show that Lilith is at least twice as fast.

In summary, it is quite feasible to conceive of a Lilith program effectively using as many as a million bytes of memory. Sixteen bits of addressing may not be quite enough for some programming wishes, but chances are that Lilith is more than adequate to handle what most people categorically assume to be too large for 16 bits.

Memory-mapped High Resolution Graphics Is Efficient

Experience with Lilith supports the idea that memory-mapped bitmap graphics is still the best way. Detractors of the idea suggest that the display refresh interferes with the speed of the processor, that the bitmap occupies valuable memory addressing space, and that a separate graphics processor can be more effective. These notions can be refuted by experience with

Lilith.

Display refresh interfered significantly with the processing power in the Alto computer, but not in Lilith. Measurements reported in chapter 6 have shown that the display refresh only slows processing by a negligible 3%. This hardly warrants a separate memory.

Use of valuable addressing space is not a problem since the bitmap can be kept in Lilith's non-prime addressing space. Therefore, there is no reason to make an external bitmap memory. If Lilith had more memory for the bitmap, it could use it just as well for other purposes than just display refresh.

The use of a separate graphics processor could be of value. Graphics processing is typically needed for intermittent intervals at a high processing rate. A processor powerful enough to perform well as a graphics processor could also be desireable as the host processor. The processor of Lilith is both a powerful graphics engine and a powerful host processor. This combination seems to be both usable and efficient. The industry has many examples of weak host processors coupled to powerful graphic engines and vice versa. Their overall performance is not satisfying. The best combination would seem to be a powerful host processor coupled with an equally powerful graphics processor. The two combined, such as in Lilith, would seem to be cost effective. Nevertheless, if a separate graphics processor is to be added, there is no reason that it could not operate out of Lilith's main memory. In fact, there is no reason it could not be a second Lilith processor.

The Weak Link in Lilith is the Microinstruction Sequencer

Measurements using the Diagnostic Processor Unit(DPU) have indicated that almost all components of Lilith could operate reliably at a rate as much as 30% faster than the current cycle time of 150 nanoseconds. The exception is the microinstruction sequencing logic which has a chain of signal propagations which will not allow this. At the heart of this chain is the AMD2911 microprogram sequencer [AM]. This circuit is either too slow, or the way it has been designed into Lilith needs to be reconsidered. Under either circumstance, this would be the place to begin for improving the Lilith's performance. The statistics reported in chapter 6 confirm this conclusion. From the distribution of execution time given in Table 6.1, one can see that an improvement of 30% in memory speed would only yield an overall performance increase of 9.2% whereas an improvement of 30% in the basic microinstruction cycle time would net an improvement of almost 21%. (This is a comparison of 30% of the total memory wait time versus 30% of the microinstruction execution time)

Measurements of the Lilith execution rate indicate that a redesign of the microinstruction sequencing logic could allow the processor to operate as much as 50% faster.

Interrupts are Delayed by Complex Lilith Instructions

Lilith enjoys a performance advantage over other processors because of its single-minded design orientation towards the execution of programs written in Modula-2. This narrow orientation has allowed the designers to identify relatively complex operations which occur frequently in Modula-2 and assign them to single microinstructions, thereby allowing Lilith to perform more work with less instruction fetch overhead. However, this advantage is lost in the case of Lilith's response to interrupts. Since Lilith can only recognize interrupts at the completion of an M-code, in some cases a particularly long M-code can delay the recognition of an interrupt more than is acceptable. In the case of some very long bitmap operations it is possible to delay an interrupt response by more than several tenths of a second.

One possible solution to this problem is to allow interrupts to be recognized during individual M-code executions. Our considerations of this alternative have led us to the conclusion that this would be difficult because of the lack of complex integrated circuit chips which would support this. Another possibility is to design peripheral interfaces with more intelligence so that longer delays in processor interrupt acknowledgement can be tolerated without loss of information. The cheapness of microcomputers is almost a proverb now, so this solution could probably be most economically realized.

Microprogrammability is an Essential Tool

We have not attempted to justify the value of Lilith's microprogrammability in this document. Therefore, this conclusion (that the ability to alter and reprogram the microcode is essential) is unsubstantiated as far as this document is concerned, but we feel it is necessary to stress the value of this feature based upon our general experience with Lilith. The ability to microprogram a computer has not been emphasized in recent years as much as it was earlier. Our opportunities for use of Lilith have been greatly expanded by virtue of the ability to develop M-codes for special applications. This facility has been used extensively to develop high speed M-codes performing very specialized functions for the laser printer. It has also been used to make higher precision floating point operations available for a computer-aided engineering application. We expect to continue to make good use of this feature.

The Diagnostic Processor is an Indispensable Aid to Debugging

We have only indirectly alluded to the value of the *diagnostic processor unit*. Therefore, this conclusion is also unsubstantiated in this document. Nonetheless, many thousands of hours spent in the pursuit of Lilith's *bugs* have convinced the author of the value of the diagnostic processor unit. It is hoped that the reader will accept this conclusion on the basis of that experience.

Co-processors Could Easily Be Added to Lilith

Having an especially elegant implementation of coroutine management such as provided by Modula-2 invites us to investigate new ways to utilize this system feature in programs. The use of coroutines as a means of logically separating a complex program into smaller, less-complex, separate coroutine tasks becomes apparent to us. Closely coupled is the idea of increasing performance by adding additional processors to process the somewhat independent computing tasks in parallel. The statistics from our analysis indicate that this could be done quite successfully. As computed in Chapter 6 with the statistics from Table 6.2, it was shown that only about a third of the Lilith's memory bandwidth was actually used by the processor in the simulated execution of the benchmark program. With two thirds of the memory bandwidth available, and four of eight available memory ports unused [Ohr], it would appear that a co-processors could easily be added to Lilith with substantial benefit. This would be especially so if one of the more powerful, and highly integrated microprocessors were selected. Or, if the additional processor were a Lilith, it is even possible that there would be additional bandwidth left to add even a third.

Figure of Merit Explains Lilith's Efficiency

If one regards data movement in memory as useful work and effort expended in fetching and decoding instructions as overhead, then Lilith shows some interesting figures of merit with regard to 1) the number of memory bytes used, 2) the number of memory cycles used, and 3) the execution time spent for each category.

First of all, from Table 6.1 the ratio of memory bytes used in the execution of our simulated program shows that Lilith has the following figure of merit:

(25,591 data memory words * 2 bytes/word)/ 32147 IFU bytes = 1.59

In chapter 6, we developed a similar figure of merit for memory cycles:

25,591 data memory cycles / 6396 IFU memory cycles = 4.00

Finally, in terms of actual execution time spent, Lilith uses 23.8% of its time for fetching and decoding instructions. The calculations are:

225 ns * 26098 M-codes + 150 ns * 6049 M-code operands
+ 1.83 ms IFU memory delay = 8.61 ms instruction fetch time

27.49 ms instruction execution / 8.61 ms inst. fetch = 3.19

In all three of the above cases, the numbers are above 1. Intuitively, we feel that this manner of measuring efficiency is accurate. However, until we have by some means measured other computers in the same fashion, we cannot be sure.

7.2 Recommendations

Based upon the measurements of the Lilith performance and experience in using it, three recommendations can be made.

Improved Microinstruction Controller Design

Experience with the design of the MCU of Lilith has shown this part of the machine to be singularly responsible for preventing a faster rate of execution. We therefore propose a faster design which can also provide the needed capabilities for *conditional jump* and *jump to subroutine* operations.

If the next microinstruction address is included in the microinstruction register, the minimum possible cycle time for a microinstruction controller would then be a function of the access time of the control store ROM, and the load time of the register as shown in Figure 7.1a.

Of course, this organization has no capability to make conditional branches, as each microinstruction has only a single possible successor. In order to add this capability, we need to add a second control store so that two microinstruction can be fetched simultaneously. Then, logic operations based on condition codes and the contents of the previous microinstruction can be used to select which of the two microinstructions will be enabled for use as the next microinstruction. The timing of this design is longer than the previous design only by the *load time* of the flipflops which enable the microinstruction register. This is shown if Figure 7.1b.

By adding a tri-state buffer which can be *gated* off, we can allow one half of the double-size control store to receive its address input from the next M-code delivered by the IFU. Now the loop timing would be increased by the propagation delay of the tri-state buffer or the delay caused by the arrival of the M-code, whichever is longer. See Figure 7.1c. Incidentally, some additional control options must be included in the microinstruction word so that the selection can be made for the right control store for receiving the next microinstruction address from the microinstruction register or from the IFU.

Interrupts are easily added at this point, by programming the interrupt vector into the next microinstruction field of the instruction word, and then adding logic to allow an interrupt to override the selection of the right MIR in favor of the left.

As a final refinement, we propose adding a small stack by way of a side path to serve as the means of performing *jump to subroutine* operations as shown in Figure 7.1d. The additional register in the loop is necessary to capture the proper return address in the stack. Note that the address of the *jump to subroutine* instruction is the address which is saved, and the instruction which is fetched upon returning from the subroutine must be the instruction from the same address in the opposite control store.

It is recognized that this microinstruction control architecture would follow an erratic path through the control store and that even some of the microinstructions might need to be duplicated in several locations. Nevertheless, we predict that these problems would be manageable in the microinstruction assembly program, and that the overall speed of operation would be significantly faster.

Improvement of Memory Addressing for Coroutines

As Lilith is currently designed, memory addresses in the first 64k words have a special status and must be shared by all tasks operating within the machine. We feel that future use of workstations will be heavily oriented towards single-users, but not necessarily towards single task operations. In other words, we see the possibility that a user may want to display several windows on the screen and initiate several independent operations through these windows. Consequently, we

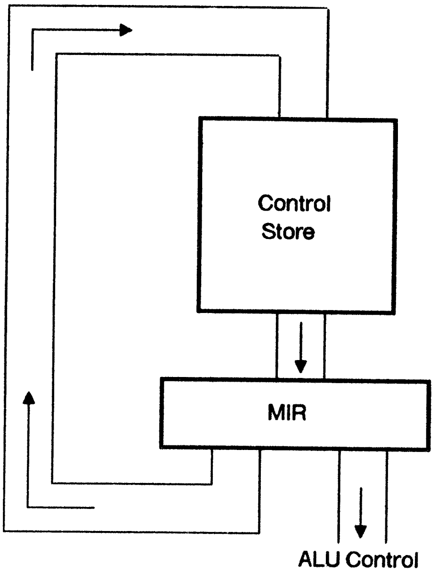


Figure 7.1a Simple MCU

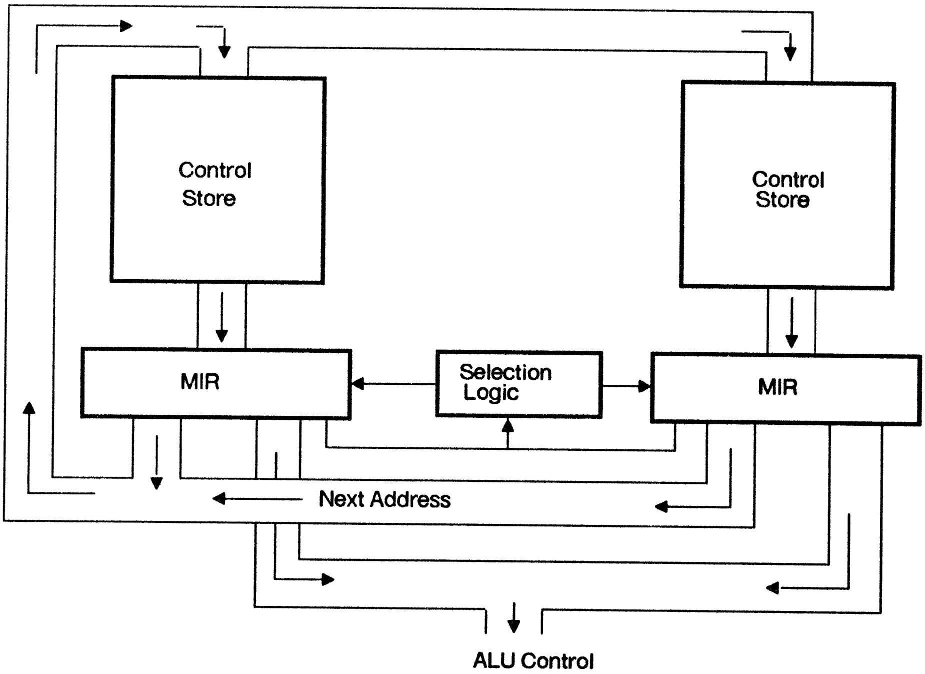


Figure 7.1b MCU with Jump Capability

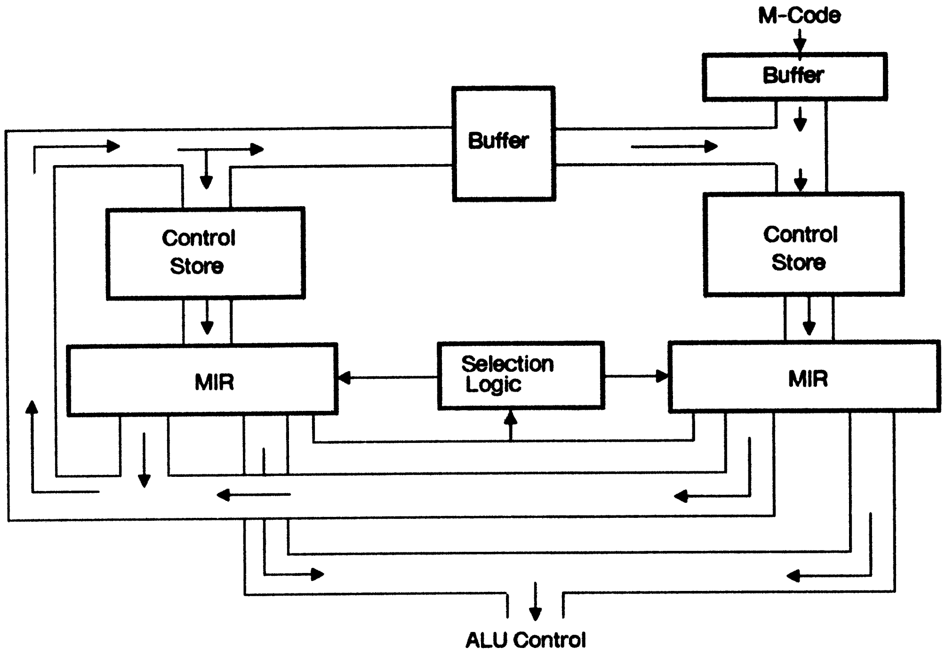


Figure 7.1c MCU with Jumps and Jump Map

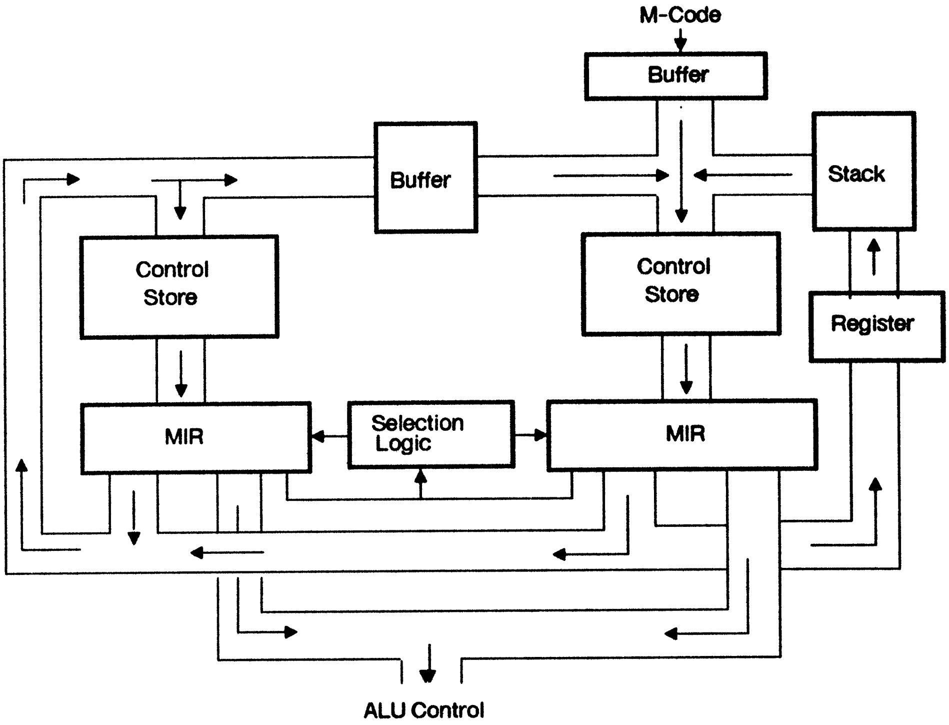


Figure 7.1d MCU with Jumps, Jump Map, and Jump Subroutine

see a need for a modification to the Lilith design which will provide an option for each task or coroutine operating within the system to have its own separate address space, uncompromised by any other. At the same time, we wish to see all shareable resources, such as code areas, shared to avoid the expense of duplication.

We think that this can be achieved quite simply. First of all, it is necessary to design a *default register* into the memory data port circuitry. The function of this register would be to provide high order page address bits for any memory operation which was intended to take place in the lower 64k words of memory. Then each coroutine could be allocated its own base page addressing space, including its own module table.

With each coroutine having its own module frame table, it would still be possible to share module code space and prevent duplicate use of memory. This would depend entirely upon the code frame pointer which is placed in the first word of the global storage area. One should note, however, that unless coroutines are allocated into the same base page of memory, global storage areas of modules will no longer be shared as before.

This leaves one problem to be resolved: how does the machine handle coroutine transfers, especially those caused by machine interrupts? We propose a solution which requires the establishment of two types of coroutine transfers: global and local. Local coroutine transfers would be those which would take place as currently implemented in the machine, without any reference to the default base page address register. Global coroutine transfers, on the other hand, would always save the value of the default base page register as well as restore such a value from the process descriptor. This implies that there be two kinds of process descriptors: global and local, as well. It would be requisite that all global process descriptors be resident in the real base page of the machine. Interrupt caused coroutine transfers would be by their very nature global transfers.

We feel that this addition to the Lilith architecture may do more to help alleviate concerns about the small addressing space of Lilith than an increased word width. At the very least, Lilith could make use of a greatly expanded memory with this addition.

As a final comment, it would be well to point out that this addition to the Lilith's architecture could possibly be completed without causing any incompatibility with current software, except those modules which have interrupt driven I/O directly programmed.

Word Size Expansion of Lilith

Many are suggesting that the processors of tomorrow need a word size greater than 16 bits. And, as the next "magical" number beyond 16, they offer 32 bits as the logical choice for the word size of the next processor. We are hesitant to espouse such a choice, for it magnifies the packaging problems of building the processor by at least a factor of four rather than two. In the future, it may be possible when memory can be economically integrated onto the same chip as the processor, but we do not find it practical or efficient now.

When the question is asked, "do we really need that much memory?", one always gets the response: "oh, but memory is cheap now." So, we propose a new processor which provides both the extra memory addressing capability (which "everyone" seems to feel is necessary), and the benefits of the smaller word size.

We propose a word size of 20 or 24 bits. Each word of memory would be of this width. Each register of the processor would also be of this width. The addressing space which could be referenced by a pointer in storage would, therefore, be 1 or 16 million words. Probably, it will be at least five years before the integration specialists can get this much memory on a single chip. Therefore, it should be adequate for now.

One can think of many objections, all related to the "unconventional" word size: the number of characters per word, the number of instruction bytes per word, the difficulty of implementing graphic operations on a bitmap constructed from 20 or 24 bit words. It is true, 16 bits has been a 'magic' number for certain operations. So, we propose to "waste" memory by continuing the orientation of the processor to 16 bits in all of these circumstances. First of all, the number of characters per word would continue to be two. Perhaps the extra bits could be used as font indicators, but not as a third character. The display interface would only "see" 16-bit words as it fetched them for the bitmap, and the graphic operators would continue to calculate bit positions with ease as they performed division by simple shifting, and modulo arithmetic with AND operations. Finally, in order to minimize the additional work needed to complete this system, the instruction set for the computer would continue to be based upon two 8-bit M-codes per word.

The only obvious objection to this machine architecture is that it wastes memory. Perhaps this objection is diminished by the advent of cheap memory.

7.3 Reflections on Lilith

It has take many years to establish the value of the Lilith architecture. Unfortunately for the hardware engineer, just building the machine is not enough to prove its value. It also takes a good deal of software effort to demonstrate the value of what has been created. The hardware technology of Lilith is already old, but the software which demonstrates its effectiveness is just emerging. Furthermore, a large amount of the software which is being developed falls under the category of software tools refinement, so that with the passage of time Lilith offers better and better software tools for the development of programs which further demonstrate its value.

One thing is clear from Lilith benchmarks: Lilith has an architectural advantage over conventional register machine designs. The amount of storage required for Lilith programs is still substantially smaller than any of the new processor designs. Also, in the benchmark comparisons, those processors offering comparable performance with Lilith are constructed from new faster technology and have cycle times which would lead one to expect them to substantially outperform Lilith. But, as an example, a 10 megahertz Motorola 68000 processor with a 100 nanosecond microcycle offers performance only approximately equal to a Lilith with a 150 nanosecond microcycle time instead of being 50 percent faster. If a Lilith were developed in chip form with a similar technological basis, it would be substantially faster.

Appendix 1 The Lilith Instruction Set

In this appendix the Lilith instruction set is presented in a clear but rather unexact form. A reader who is relatively unexperienced in the structure of stack machines and in the use of the Modula-2 programming language may find the information as it is presented here easier to digest. However, the most accurate and the official definition for the Lilith M-code machine operation is defined by the pseudo-modula-2 program, known as the "interpreter", which has been published in documents by Wirth [Wir80] and Jacobi [Jac], and in the Lilith Hardware Manual [Ohr]. If any ambiguity arises in understanding how an instruction presented here operates, it should be resolved by reference to this program.

For use in understanding the instruction set as it is presented here, the following notations must be explained.

a) The relative sizes of the boxes denotes the number of bits used. The smallest size is 4 and the most frequently used size is 8. The largest size is a word, i.e. 16 bits.

b) The term 'tos' means 'top of the expression evaluation stack'.

c) The terms 'push', 'pop', and 'load' are always used with respect to moving data into and out of the expression evaluation stack.

d) Using a value as an 'address' always implies access into main memory.

e) The letters "P", "G", "L", "S", "PC", "F", "M", and "H" refer to the registers which have been described in Chapter 3.

f) The value of the program counter, "PC", always 'points' to the next byte after the opcode portion of the instruction.

Load Immediate Instructions

LI	n		Push n
LI	n		Push n
LIW	n		Push n
LID	m	n	Push m; Push n
LIN			Push NIL

Load Address Instructions

LLA	n		Push address of word addressed by L + n
LGA	n		Push address of word addressed by G + n
LSA	n		Replace tos with tos + n
LEA	m	n	Push address of nth word in global storage of module m

Jump Instructions

JPC	n	Jump to PC + n if tos = 0; pop tos
JP	n	Jump to PC + n
JPFC	n	Jump to PC + n if tos = 0; pop tos
JPF	n	Jump to PC + n
JPBC	n	Jump to PC - n if tos = 0; pop tos
JPB	n	Jump to PC - n
ORJP	n	Jump to PC + n if tos <> 0 else increment PC and pop
ANDJP	n	Jump to PC + n and push 0 else increment PC and pop

Load Local

LLW	n		Push word addressed by L + n
LLW	n		Push word addressed by L + n
LLD	n		Push double word addressed by L + n

Store Local

SLW	n		Pop word into word addressed by L + n
SLW	n		Pop word into word addressed by L + n
SLD	n		Pop double word into double word addressed by L + n

Load External

LEW	m	n	Push nth word in global storage of module m
LED	m	n	Push double word at nth word of global storage of module m

Store External

SEW	m	n	Pop word into nth word of global storage of module m
SED	m	n	Pop double word into nth word of global storage of module m

Load Global

LGW	n		Push word addressed by G + n
LGW	n		Push word addressed by G + n
LGD	n		Push double word addressed by G + n

Store Global

SGW	n		Pop word into word addressed by G + n
SGW	n		Pop word into word addressed by G + n
SGD	n		Pop double word into double word addressed by G + n

Load Stack

LSW | n	Replace tos with word addressed by tos + n
LSW | n	Replace tos with word addressed by tos + n
LSD | n	Push double word addressed by tos + n; (tos removed)
LSD0	Push double word addressed by tos; (tos removed)

Store Stack

SSW | n	Store tos in word addressed by tos-1 + n (tos, tos-1 removed)
SSW | n	Store tos in word addressed by tos-1 + n (tos, tos-1 removed)
SSD | n	Store tos double word in double word addressed by tos-2 + n; (tos, tos-1, tos-2 removed)
SSD0	Store tos double word in double word addressed by tos-2; (tos, tos-1, tos-2 removed)

Load Indexed

LXB	Get word addressed by tos/2 + tos-1; If tos is even then push left byte else push right byte (tos, tos-1 removed)
LXW	Push word addressed by tos + tos-1; (tos, tos-1 removed)
LXD	Push the double word addressed by tos + tos-1; (tos, tos-1 removed)
LXFW	Push the word addressed by tos + tos-1 • 4; (tos, tos-1 removed)

Store Indexed

SXB	Compute word address of tos/2 + tos-1; If tos is even then store tos-2 in left half of word address else store tos-2 in right half (tos, tos-1, tos-2 removed)
SXW	Store tos in word addressed by tos-1 + tos-2 (tos, tos-1, tos-2 removed)
SXD	Store tos double word in double word addressed by tos-2 + tos-3; (tos, tos-1, tos-2, tos-3 removed)
SXFW	Store tos in word addressed by tos-1 + tos-2 • 4 (tos, tos-1, tos-2 removed)

Unsigned Boolean Compares

ULSS	If tos < tos-1 Push 1 else Push 0; (tos, tos-1 removed)
ULEQ	If tos <= tos-1 Push 1 else Push 0; (tos, tos-1 removed)
UGTR	If tos > tos-1 Push 1 else Push 0; (tos, tos-1 removed)
UGEQ	If tos >= tos-1 Push 1 else Push 0; (tos, tos-1 removed)
EQL	If tos = tos-1 Push 1 else Push 0; (tos, tos-1 removed)
NEQ	If tos <> tos-1 Push 1 else Push 0; (tos, tos-1 removed)

Signed Boolean Compares

LSS	If tos < tos-1 Push 1 else Push 0; (tos, tos-1 removed)
LEQ	If tos <= tos-1 Push 1 else Push 0; (tos, tos-1 removed)
GTR	If tos > tos-1 Push 1 else Push 0; (tos, tos-1 removed)
GEQ	If tos >= tos-1 Push 1 else Push 0; (tos, tos-1 removed)

Dyadic Boolean Operations

OR	Push tos OR tos-1 (tos, tos-1 removed)
XOR	Push tos XOR tos-1 (tos, tos-1 removed)
AND	Push tos AND tos-1 (tos, tos-1 removed)

Signed Unary Operation

ABS	Push absolute value of tos (tos removed)
NEG	Push 2's complement of tos (tos removed)
COM	Push 1's complement of tos (tos removed)

Unsigned Dyadic Operation

UADD	Push tos + tos-1 (tos, tos-1 removed)
USUB	Push tos-1 - tos (tos, tos-1 removed)
UMUL	Push tos * tos-1 (tos, tos-1 removed)
UDIV	Push tos-1 / tos (tos, tos-1 removed)
UMOD	Push tos-1 MOD tos (tos, tos-1 removed)

Signed Dyadic Operation

ADD	Push tos + tos-1 (tos, tos-1 removed)
SUB	Push tos-1 - tos (tos, tos-1 removed)
MUL	Push tos * tos-1 (tos, tos-1 removed)
DIV	Push tos-1 / tos (tos, tos-1 removed)

Range Checking

UCHK	Unsigned: Rangecheck trap if tos-2 < tos-1 or tos-2 > tos; (tos, tos-1 removed)
CHK	Signed: Rangecheck trap if tos-2 < tos-1 or tos-2 > tos; (tos, tos-1 removed)
CHKZ	Rangecheck trap if tos-1 > tos (tos removed)
CHKS	Rangecheck trap if tos < 0

Bit Test

IN	If tos-1 IN tos Push1 else Push 0 (tos,tos-1 removed)

Test and Set

TS	Push word addressed by tos; Set word addressed by tos to 1 (remove tos)

Load String Address

LSTA	n	Push n + word addressed by G + 2

Stack Operations

ENTR n	Set S to S + n
ALLOC	Push S; Set S to S + tos; (tos removed)
STOT	Copy tos to Procedure Stack; (tos removed)
COPT	Push tos
DECS	Decrement S
PCOP n	Word addressed by L + n set to S; Move tos words starting at tos-1 into word addressed by S; Set S to S + tos; (tos, tos-1 removed)

Procedure Operations

CX m n	Call the nth procedure in module m
CI n	Call the nth local procedure
CF	Get the word on the top of the procedure stack; Call the procedure of the right half of the word, in the module of the left half of the word
CL n	Call the nth local procedure
CL n	Call the nth local procedure
RTN	Return from Procedure
GB n	Get Base address (static link) n levels down
GB1	Get Base address one level down

Appendix 2 The Program "Simulator"

"Simulator," a program written in Modula-2, simulates as closely as possible the execution of microinstructions in the Lilith computer. The purpose in writing such a program is to provide a means to discover the dynamic nature of the processor and thereby answer such questions regarding the usage of memory, the effectiveness of the instruction fetch unit, or the dynamic occurrence of *wait for memory* condition.

The organization of Simulator is identical to the function organization of the Lilith. However, the functional components of Simulator are programs modules, whereas the function components of the Lilith are circuit boards or portions of circuit boards.

The modules of Simulator (bearing identical names to the functional components of the Lilith) are interconnected by IMPORT and EXPORT statements. These interconnections parallel the Lilith interconnections found in the back plane. Quite often, segments of the Simulator programming exactly reflect the electronic components used in the actual circuitry, and other times the programming takes a circuitous route to achieve the correct functional simulation of the hardware. The arithmetic logic unit, for example, required some intricate programming in order to correctly sense the carry, overflow and other condition codes. Such deviations from exact simulation are, however, confined to the execution of a single microinstruction, so that the various registers of the machine always contain exactly the correct data at the end of each microinstruction.

The correctness of Simulator was established in an interesting fashion. Since the Lilith is already equipped with a diagnostic processor which can load programs and execute them in a single instruction mode fashion, it was possible to debug the Simulator by comparing its results to the results reported by the diagnostic processor. A special terminal emulator program was used for this purpose. The emulator alternately requests single instruction execution results from the diagnostic processor and computes simulated microinstruction executions through procedure calls to Simulator. The speed of this comparison is slow, less than a hundred microinstructions per second. Nevertheless, it is fast enough to enable almost complete testing of the program Simulator. Ironically, this technique portends to be of real value in isolating faults in malfunctioning machines.

Several files are required as data input for Simulator. These are files for the control store ROM's, the map ROM's, and the boot program. They are output files from the microinstruction assembler and from the "Bootlink" program. Simulator uses these files unmodified.

```
MODULE Simulator;
  (* R. Ohran, Nov, 1983*)
IMPORT SYSTEM;
IMPORT FileSystem;
IMPORT Terminal;
IMPORT WinDisk;
FROM CardinalIO IMPORT WriteOct;
FROM Terminal IMPORT WriteLn,WriteString,Read,ReadAgain,Write;

CONST
  bdALU  = 0;  bsALU     = 0;
  bdMD   = 1;  bsMD      = 1;
  bdPC   = 2;  bsPC      = 2;
  bdMAR  = 3;  bsIR4     = 3;
  bdSR   = 4;  bsIR8     = 4;
  bdMDS  = 5;  bsIRMINUS = 5;
  bdIOA  = 6;  bsIRSTAR  = 6;
  bdIOD  = 7;  bsIOD     = 7;
  bdINM  = 8;  bsINM     = 8;

  bdBNK  =13;
  bdF    =14;  bsF       =14;
  bdDASH =15;  bsDASH    =15;
               bsINT     =16;

  oneCycle = 1;
  oneCode = -1;

TYPE MicroInstruction = ARRAY[0..4] OF CHAR;

VAR uad:CARDINAL;                     (*next microinstruction address*)
    bus:CARDINAL;                     (*processor bus*)
    mir: MicroInstruction;            (*microinstruction register*)
    iMask:CARDINAL;                   (*interrupt mask register*)
    ioAddress:CARDINAL;               (*I/O address register*)
    i,d,ucnt,ecnt:CARDINAL;
    aluinstr:BOOLEAN;
    interrupt:BOOLEAN;
    ch:CHAR;
    time:CARDINAL;                    (*counts time in nanoseconds*)
    mcnt:ARRAY [1..20] OF CARDINAL;   (*counts microinstructions in M-codes*)

PROCEDURE WriteState;
BEGIN
       WriteOct(time);time:=0;
       WriteOct(Memory.readcnt);Memory.readcnt:=0;
       WriteOct(Memory.writecnt);Memory.writecnt:=0;
       WriteOct(IFU.ifubytes);IFU.ifubytes:=0;
       WriteOct(IFU.memcnt);IFU.memcnt:=0;
       WriteOct(IFU.mcodecnt);IFU.mcodecnt:=0;
       WriteOct(MIR.uinstcnt);MIR.uinstcnt:=0;
       WriteOct(IFU.wait);IFU.wait:=0;
       WriteOct(MDP.wait);MDP.wait:=0;
       WriteOct(MIR.testinstcnt);MIR.testinstcnt:=0;
       WriteLn;
       WriteOct(ALU.pushcnt);ALU.pushcnt:=0;
       WriteOct(ALU.popcnt);ALU.popcnt:=0;
       WriteOct(ALU.opcnt);ALU.opcnt:=0;
```

```
        WriteOct(IFU.mcodedelay);IFU.mcodedelay:=0;
        WriteOct(IFU.delay1);IFU.delay1:=0;
        WriteOct(IFU.waitcnt);IFU.waitcnt:=0;
        WriteOct(ALU.qcount);ALU.qcount:=0;
        WriteOct(ecnt);ecnt:=0;
        WriteOct(jsrcnt);jsrcnt:=0;
      WriteLn;
END WriteState;

MODULE HPHEX;
  (* implements a formatted input/output to the HPterminal *)
  IMPORT Write, Read, WriteLn, WriteString,ReadAgain;
  EXPORT hexin, hexout,hexbyteout,hex12out;

  PROCEDURE hexout(n : CARDINAL);
    VAR help1, help2:CARDINAL;
    BEGIN help1 := 1000h;
      REPEAT help2 := (n DIV help1) MOD 16;
        help1:= help1 DIV 16;
        IF help2 <= 9 THEN Write( CHAR(help2+30h))
                      ELSE Write( CHAR(help2+37h))
        END
      UNTIL help1 = 0;
    END hexout;

  PROCEDURE hexbyteout(n : CARDINAL);
    VAR help1, help2:CARDINAL;
    BEGIN help1 := 10h;
      REPEAT help2 := (n DIV help1) MOD 16;
        help1:= help1 DIV 16;
        IF help2 <= 9 THEN Write( CHAR(help2+30h))
                      ELSE Write( CHAR(help2+37h))
        END
      UNTIL help1 = 0;
    END hexbyteout;

  PROCEDURE hex12out(n : CARDINAL);
    VAR help1, help2:CARDINAL;
    BEGIN help1 := 100h;
      REPEAT help2 := (n DIV help1) MOD 16;
        help1:= help1 DIV 16;
        IF help2 <= 9 THEN Write( CHAR(help2+30h))
                      ELSE Write( CHAR(help2+37h))
        END
      UNTIL help1 = 0;
    END hex12out;

  PROCEDURE hexin(VAR n:CARDINAL);
    VAR ch:CHAR; help:CARDINAL;
    BEGIN n:=0;
      Read(ch);Write(ch);
      WHILE ((ch>='0')AND(ch<='9')) OR ((ch>='A') AND (ch<='F')) DO
        IF ch <= '9' THEN help:=CARDINAL(ch)-30h
                     ELSE help:=CARDINAL(ch)-37h
        END;
        n:=n*16+help;
        Read(ch);
      END;
      ReadAgain;
    END hexin;
END HPHEX;

MODULE IO;
FROM SYSTEM IMPORT WORD;
FROM  WinDisk   IMPORT DiskRead,DiskWrite;
FROM  FileSystem IMPORT Response;
FROM Terminal IMPORT BusyRead,Write;
IMPORT bus,uad,ioAddress,interrupt,WriteState;
IMPORT WriteOct,time;
EXPORT QUALIFIED Get,Put;

CONST secno = 3B;
      cyllo = 4B;
      cylhi = 5B;
      sdh = 6B;

VAR lastuad,lastIOData,bindex,sector:CARDINAL;
    ch:CHAR;r:Response;
    buffer:RECORD
      CASE BOOLEAN OF
        TRUE:c:ARRAY[0..255] OF CHAR|
        FALSE:w:ARRAY[0..127] OF WORD
      END;
     END;
    diskreg:ARRAY[0..7] OF CARDINAL;

    PROCEDURE GET(channel: CARDINAL; VAR info: WORD);
      (* input from channel to info *)
    CODE 240B (* Read *)
    END GET;

    PROCEDURE PUT(channel: CARDINAL; info: WORD);
      (* output info to channel *)
    CODE 241B (* Write *)
    END PUT;

PROCEDURE Get;
BEGIN
  IF (lastuad + 1) = uad THEN
  ELSIF ioAddress  = 20B THEN
     lastIOData:=CARDINAL(buffer.c[bindex]);bindex:=(bindex + 1) MOD 256;
  ELSIF ioAddress = 1 THEN BusyRead(ch);
    IF ch = 0C THEN lastIOData:=0 ELSE lastIOData:=1 END;
  ELSIF ioAddress = 2 THEN lastIOData:=CARDINAL(ch)
  ELSE GET(ioAddress,lastIOData)
  END;
  bus:=lastIOData;
  lastuad:=uad
END Get;

PROCEDURE Put;
BEGIN
    IF (lastuad + 1) = uad THEN
    ELSIF (ioAddress DIV 8) = 2 THEN
      IF ioAddress = 20 THEN buffer.c[bindex]:=CHAR(bus);
        INC(bindex);
        IF bindex>=256 THEN (*write sector possibly*)
```

```
          bindex:=0 END;
      ELSIF ioAddress = 27B THEN
        IF bus = 20H THEN
          sector:= diskreg[secno] MOD 32 + 32*(diskreg[sdh] MOD 2)
                  +diskreg[cyllo]*64 + diskreg[cylhi]*256*64;
          DiskRead(0,sector,buffer.w,r);bindex:=0;
        ELSIF bus= 30H THEN (*disk write command*) bindex:=0;
        ELSE PUT(ioAddress,bus) END;
      ELSE diskreg[ioAddress MOD 8]:= bus END
    ELSIF ioAddress = 3 THEN interrupt:=FALSE
    ELSIF ioAddress = 0 THEN Write('#');WriteState;
    ELSE PUT(ioAddress,bus) END;
    lastuad:=uad
END Put;
BEGIN
  lastuad:=0;
END IO;

MODULE ALU;
IMPORT MIR;
IMPORT bus;
IMPORT bdMDS,bdALU,bsIR4;
EXPORT QUALIFIED ccCodes,Reg,RAMA,y,popcnt,pushcnt,opcnt,
                 ClockALU,shiftCountReg,EvaluateALU,
                 qloaded,qcount,lastqcnt;

CONST
(*  symbolic  "I" field names      *)
(*  I0-I3      I3-I5      I6-I8    *)

    AQ = 0;   ADD = 0;  QREG = 0; R = 0;
    AB = 1;  SUBR = 1;   NOP = 1; C = 1;
    ZQ = 2;  SUBS = 2;  RAMA = 2; Z = 2;
    ZB = 3;    Or = 3;  RAMF = 3; V = 3;
    ZA = 4;   And = 4; RAMQD = 4; F = 4;
    DA = 5; NOTRS = 5;  RAMD = 5; S = 5;
    DQ = 6;  EXOR = 6; RAMQU = 6; H = 6;
    DZ = 7; EXNOR = 7;  RAMU = 7; E = 7;

VAR
    stkindex,stkcnt,dinputs,y,f:CARDINAL;
    shiftCountReg,popcnt,pushcnt,opcnt:CARDINAL;
    Qreg ,qcount,lastqcnt         : CARDINAL;
    qloaded:BOOLEAN;
    Reg : ARRAY [0..15] OF CARDINAL;
    stack: ARRAY [0..15] OF CARDINAL;
    newccCodes,ccCodes: BITSET;

PROCEDURE DADD(xh,hl,yh,yl:CARDINAL):REAL;
CODE 210B;
END DADD;

PROCEDURE SHM(mask:CARDINAL;ch,n:CARDINAL):CARDINAL;
CODE 275B;322B;
END SHM;

PROCEDURE EvaluateALU;
VAR r,s,carry:CARDINAL;
    trick:RECORD
```

```
            CASE BOOLEAN OF
              FALSE:  H,L:CARDINAL|
              TRUE:   real:REAL
            END
          END;
BEGIN
  EvaluateShifter;

  IF (MIR.bd = bdMDS) THEN
   MIR.rs:= CARDINAL(BITSET(MIR.rs) * {13,15} );
   IF  NOT (14 IN BITSET(MIR.dest)) THEN
     IF NOT(15 IN BITSET(Qreg)) THEN INC(MIR.rs,2) END;
   ELSIF C IN ccCodes THEN INC(MIR.rs,2) END;
  END;
  lastqcnt:=qcount;
  CASE MIR.rs OF
    AQ : r:=Reg[MIR.a];s:=Qreg; IF qloaded THEN INC(qcount) END|
    AB : r:=Reg[MIR.a];s:=Reg[MIR.b];|
    ZQ : r:= 0;s:=Qreg;IF qloaded THEN INC(qcount) END|
    ZB : r:= 0;s:=Reg[MIR.b];|
    ZA : r:= 0;s:=Reg[MIR.a];|
    DA : r:= dinputs;s:=Reg[MIR.a];|
    DQ : r:= dinputs;s:=Qreg;IF qloaded THEN INC(qcount) END|
    DZ : r:= dinputs;s:=0;
  END; (* rs CASE *)

  CASE MIR.c OF
0: carry:=CARDINAL(C IN ccCodes)|
1: carry:=CARDINAL(NOT (C IN ccCodes))|
2: carry:=1|
3: carry:=0
  END;

  newccCodes:= {};

  CASE MIR.fct OF
   ADD    : trick.real:=DADD(0,r,0,s);
            IF BOOLEAN(carry) THEN
              trick.real:=DADD(trick.H,trick.L,0,1) END;
            f:=trick.L;
            IF 15 IN BITSET(trick.H) THEN INCL(newccCodes,C) END;
            IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
            IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
            IF f=0 THEN INCL(newccCodes,Z) END;
            IF BOOLEAN(BITSET(SHM(1,CARDINAL(BITSET(f)/BITSET(r)/
             BITSET(s)),15))/BITSET(trick.H)) THEN INCL(newccCodes,V) END;|

  SUBR    : r:=0FFFFH - r;
            trick.real:=DADD(0,r,0,s);
            IF BOOLEAN(carry) THEN
              trick.real:=DADD(trick.H,trick.L,0,1) END;
            f:=trick.L;
            IF 15 IN BITSET(trick.H) THEN INCL(newccCodes,C) END;
            IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
            IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
            IF f=0 THEN INCL(newccCodes,Z) END;
            IF BOOLEAN(BITSET(SHM(1,CARDINAL(BITSET(f)/BITSET(r)/
             BITSET(s)),15))/BITSET(trick.H)) THEN INCL(newccCodes,V) END;|
   SUBS  : s:=0FFFFH - s;
           trick.real:=DADD(0,r,0,s);
           IF BOOLEAN(carry) THEN
             trick.real:=DADD(trick.H,trick.L,0,1) END;
           f:=trick.L;
           IF 15 IN BITSET(trick.H) THEN INCL(newccCodes,C) END;
           IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
           IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
           IF f=0 THEN INCL(newccCodes,Z) END;
           IF BOOLEAN(BITSET(SHM(1,CARDINAL(BITSET(f)/BITSET(r)/
            BITSET(s)),15))/BITSET(trick.H)) THEN INCL(newccCodes,V) END;|

     Or  : f := CARDINAL(BITSET(r) + BITSET(s));
           IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
           IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
           IF f=0 THEN INCL(newccCodes,Z) END;|

    And  : f := CARDINAL(BITSET(r) * BITSET(s));
           IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
           IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
           IF f=0 THEN INCL(newccCodes,Z) END;|

   NOTRS : f := CARDINAL((BITSET(r)/{0..15}) * BITSET(s));
           IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
           IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
           IF f=0 THEN INCL(newccCodes,Z) END;|

   EXOR  : f := CARDINAL(BITSET(r) / BITSET(s));
           IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
           IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
           IF f=0 THEN INCL(newccCodes,Z) END;|

   EXNOR : f := CARDINAL(BITSET(r) / BITSET(s) / {0..15});
           IF 8 IN BITSET(f) THEN INCL(newccCodes,H) END;
           IF 0 IN BITSET(f) THEN INCL(newccCodes,F) END;
           IF f=0 THEN INCL(newccCodes,Z) END;

  END; (* fct CASE *)

  IF V IN newccCodes THEN
    IF NOT(F IN newccCodes) THEN INCL(newccCodes,S) END
  ELSIF F IN newccCodes THEN INCL(newccCodes,S)
  END;

  IF stkindex= 0 THEN INCL(newccCodes,E) END;

  IF MIR.dest = RAMA THEN y:= Reg[MIR.a] ELSE y:=f END;

END EvaluateALU;

PROCEDURE ClockALU;

BEGIN
  CASE MIR.dest OF
    QREG   : Qreg := f ;IF MIR.bs <> bsIR4 THEN qloaded:=FALSE END;|
    NOP    : |
    RAMA   : Reg[MIR.b]:= f  |
    RAMF   : Reg[MIR.b]:= f  |
    RAMQD  : Reg[MIR.b]:= f DIV 2 + SHM(8000H,CARDINAL(C IN newccCodes),1);
```

```
          Qreg:= Qreg DIV 2 + SHM(8000H,f,1);qloaded:=FALSE |
   RAMD  : Reg[MIR.b]:= f DIV 2 + SHM(8000H,CARDINAL(C IN newccCodes),1);
 |
   RAMQU : Reg[MIR.b]:= (f MOD 32768) * 2 + SHM(1,Qreg,15);
          Qreg:= (Qreg MOD 32768)* 2 + CARDINAL(C IN ccCodes);
          qloaded:=FALSE |
   RAMU  : Reg[MIR.b]:= (f MOD 32768) * 2 + SHM(1,Qreg,15)
  END; (* dest CASE *)

  IF (MIR.bd = bdALU) AND NOT MIR.s THEN stack[stkindex]:=y END;
  IF NOT MIR.s THEN
    IF MIR.bd = bdALU THEN stkcnt:=(stkcnt +15) MOD 16;INC(pushcnt);
    ELSE stkcnt := (stkcnt + 1 ) MOD 16;
      IF MIR.dest = RAMA THEN INC(popcnt) ELSE INC(opcnt) END;
    END;
  END;
  ccCodes:=newccCodes;
 END ClockALU;

 PROCEDURE EvaluateShifter;

  PROCEDURE ROR(data,shift:CARDINAL):CARDINAL;
    CODE 275B
  END ROR;

 BEGIN
  IF (MIR.bd = bdALU) AND NOT MIR.s THEN stkindex:=(stkcnt + 15) MOD 16
     ELSE stkindex:= stkcnt END;
  IF MIR.bd = bdALU THEN
    IF MIR.sh > 1 THEN
      dinputs:= ROR(bus,shiftCountReg)
    ELSE dinputs := ROR(bus,MIR.shff)'
    END;
    IF ODD(MIR.sh) THEN
      IF MIR.sh > 1 THEN dinputs:= CARDINAL(BITSET(dinputs)
                  + BITSET(ROR(1,shiftCountReg)-1))
      ELSE dinputs:= CARDINAL(BITSET(dinputs)+ BITSET(ROR(1,MIR.shff)-1))
      END;
    END;
    IF MIR.bs = bsIR4 THEN dinputs:= dinputs MOD 16 END

  ELSE dinputs:= stack[stkindex]
  END
 END EvaluateShifter;

BEGIN
  stkcnt:=0;
  shiftCountReg:=0;
  popcnt:=0;
  pushcnt:=0;
  opcnt:=0;
  qcount:=0;
  qloaded:=FALSE
END ALU;

MODULE MCU;
IMPORT MIR;
IMPORT ALU;
IMPORT bus,uad;
IMPORT bsIR4,bsINT;
FROM Terminal IMPORT WriteString,WriteLn;
FROM FileSystem IMPORT File,Lookup,Response,ReadChar;

EXPORT NextUAD,jsrcnt;

CONST
pcPOP  = 0;
pcRTN  = 1;
pcJSR  = 3;
pcDASH = 4;
pcLOP  = 5;
pcJMP  = 7;
clockVector = 12;

VAR cc:BOOLEAN;
    mux2911,uindex,jsrcnt:CARDINAL;
    ustack:ARRAY[0..3] OF CARDINAL;
    mapROM:ARRAY[0..255] OF CARDINAL;

PROCEDURE NextUAD;
BEGIN
  IF MIR.k THEN mux2911:= MIR.pc
    ELSE
      cc:= NOT((MIR.ccMask * ALU.ccCodes) = {});
      IF (NOT MIR.ccSense AND cc) OR (MIR.ccSense AND NOT cc) THEN
        mux2911:= MIR.pc
      ELSE mux2911:= pcDASH END;
    END;
  CASE mux2911 OF
  pcJMP: IF (MIR.bs = bsIR4) THEN
           uad:= mapROM[bus]
         ELSIF (MIR.bs = bsINT) THEN
           uad:= clockVector
         ELSE uad:=MIR.jadr
         END |
  pcJSR: ustack[uindex]:=uad+1;
         uad:=MIR.jadr; uindex:=(uindex + 1) MOD 4 ;
         INC(jsrcnt)|
  pcRTN: uindex:=(uindex + 3) MOD 4;
         uad:=ustack[uindex] |
  pcDASH: uad:=uad+1 |
  pcLOP: uad:= ustack[(uindex + 3) MOD 4] |
  pcPOP: uad:=uad+1; uindex:=(uindex + 3) MOD 4
  ELSE WriteString('bad uad error') END;
END NextUAD;

PROCEDURE ReadMapROM;
VAR f:File;
    c:CHAR;
    tmp,i:CARDINAL;
BEGIN
  Lookup(f,'DK.int13dot2.MAP',FALSE);
    IF f.res = notdone THEN
      WriteLn;WriteString('int13dot2.MAP not found.');
      HALT;
    END;
  FOR i:=0 TO 47H DO ReadChar(f,c); END;
  FOR i:=0 TO 127 DO
    ReadChar(f,c);
    tmp:=CARDINAL(c)*16;
    ReadChar(f,c);
    mapROM[2*i]:=tmp+(CARDINAL(c) DIV 16);
    tmp:=(CARDINAL(c) MOD 16)*256;
    ReadChar(f,c);
    mapROM[2*i+1]:=tmp+CARDINAL(c);
  END;
END ReadMapROM;

BEGIN
  ReadMapROM;
  uindex:=0;
  jsrcnt:=0
END MCU;

MODULE MIR;
IMPORT bus,uad,mir,MicroInstruction,aluinstr;
FROM FileSystem IMPORT File,Lookup,Close,Response,Reset,ReadChar;
FROM Terminal IMPORT WriteLn, WriteString;

EXPORT QUALIFIED LoadMIR,
      dest,fct,rs,c,a,b,sh,pc,sp,k,s,e,shff,
      bd,bs,n,jadr,ccSense,ccMask,uinstcnt,testinstcnt;

VAR dest,fct,rs,c,a,b,sh,pc,shff,bd,bs,n,jadr,
    uinstcnt,testinstcnt: CARDINAL;
    sp,k,s,e,ccSense:BOOLEAN;
    ccMask:BITSET;
    controlstore:ARRAY[0..4095] OF MicroInstruction;

PROCEDURE SHM(mask:CARDINAL;ch:CHAR;n:CARDINAL):CARDINAL;
CODE 275B;322B;
END SHM;

PROCEDURE LoadMIR;
BEGIN
  mir:=controlstore[uad];
  jadr:=SHM(0FF0H,mir[4],12) + SHM(0FH,mir[3],4);
  dest:= SHM(7,mir[4],5);
  fct := SHM(7,mir[4],2);
  rs:= SHM(6,mir[4],15) + SHM(1,mir[3],7);
  c := SHM(3,mir[3],5);
  a := SHM(15,mir[3],1);
  b:= SHM(8,mir[3],13) + SHM(7,mir[2],5);
  sh:= SHM(3,mir[2],3);
  pc:= CARDINAL(mir[2]) MOD 8;
  sp:= 8 IN BITSET(mir[1]);
  k := 9 IN BITSET(mir[1]);aluinstr:=k;
  s := 10 IN BITSET(mir[1]);
  e := 11 IN BITSET(mir[1]);
  shff := CARDINAL(mir[1]) MOD 16;
  bd := SHM(15,mir[0],4);
  bs := CARDINAL(mir[0]) MOD 16;
  n := CARDINAL(mir[0]);
  ccMask := BITSET(SHM(1F00H,mir[2],11) + SHM(0E000H,mir[3],3));
  ccSense := 12 IN BITSET(mir[3]);
  INC(uinstcnt);
  IF NOT k THEN INC(testinstcnt) END;
END LoadMIR;

PROCEDURE LoadROMfromFiles;

  PROCEDURE ReadFileAndStoreInROM(fileName:ARRAY OF CHAR; index:CARDINAL);
  VAR i,tmp,lineLength,sum,address,highAddressByte : CARDINAL;
      c : CHAR;
      f : File;

    PROCEDURE ReadByte():CARDINAL;
    VAR byte : [0..255];

      PROCEDURE ReadHex():CARDINAL;
      VAR c : CHAR;
      BEGIN
        c:=0C;
        WHILE (c<'0') DO ReadChar(f,c);END;
        IF c >='A' THEN
          RETURN (ORD(c)-ORD('A')+10);
        ELSE
          RETURN (ORD(c)-ORD('0'));
        END;
      END ReadHex;

    BEGIN
      byte:=ReadHex()*16;
      RETURN(CARDINAL(byte+ReadHex()));
    END ReadByte;

  BEGIN
    Lookup(f,fileName,FALSE);
    IF f.res = notdone THEN
      WriteLn;WriteString(fileName);WriteString(' not found.');
      HALT;
    END;
    sum:=0; address:=0;
    FOR i:=1 TO 11 DO ReadChar(f,c); END; (* delete header *)
    LOOP
      ReadChar(f,c);
      IF c <>'S' THEN
        WriteString(' S missing at head of line ');
        HALT;
      END;
      ReadChar(f,c);
      IF c = '9' THEN EXIT; ELSIF c <> '1' THEN
        WriteString(' S1 missing at head of line');
        HALT;
      END;
      lineLength := ReadByte()-3;
      highAddressByte := ReadByte();
      tmp:=highAddressByte*256+ReadByte();
      IF address <> tmp THEN
        address:=tmp;
      END;
      FOR i:=1 TO lineLength DO
        tmp := ReadByte();
        controlstore[address][index]:= CHAR(tmp);
        INC(address);
      END; (* FOR *)
      ReadChar(f,c);ReadChar(f,c);ReadChar(f,c);
      IF c <> 36C THEN WriteString(' EOL missing '); HALT;END;
      sum:=0;
    END; (* LOOP *)
    Close(f);
  END ReadFileAndStoreInROM;

BEGIN
  ReadFileAndStoreInROM('DK.int13dot2.BM1 ',0);
  ReadFileAndStoreInROM('DK.int13dot2.BM2 ',1);
  ReadFileAndStoreInROM('DK.int13dot2.BM3 ',2);
  ReadFileAndStoreInROM('DK.int13dot2.BM4 ',3);
  ReadFileAndStoreInROM('DK.int13dot2.BM5 ',4);
END LoadROMfromFiles;

BEGIN
LoadROMfromFiles;uinstcnt:=0;testinstcnt:=0;
END MIR;

MODULE IFU;
IMPORT bus,Memory,MDP,ALU;
FROM FileSystem IMPORT File,Lookup,Response,ReadChar,Close;
FROM Terminal IMPORT Write,WriteString,WriteLn;
EXPORT QUALIFIED BSIR8,BSIR4,BSIRSTAR,BSIRMINUS,LoadPC,LoadF,f,pc,
      memcnt,ifubytes,wait,delay,mcodecnt,idle,delay1,mcodedelay,waitcnt;

VAR pc,f,fhi,flo,memcnt,ifubytes,wait,delay:CARDINAL;
    mcodedelay,mcodecnt,idle,delay1,waitcnt:CARDINAL;
    boot:BOOLEAN;
    reg:RECORD
      CASE BOOLEAN OF
        TRUE: words: ARRAY[0..3] OF CARDINAL|
        FALSE: bytes: ARRAY[0..7] OF CHAR
      END
    END;
    bootROM:ARRAY[0..2047] OF CHAR;

PROCEDURE IFUDelay;
BEGIN
  IF delay>0 THEN INC(waitcnt) END;
  IF MDP.delay > delay
  THEN DEC(MDP.delay,delay);
  ELSIF MDP.delay>0 THEN INC(MDP.wait,MDP.delay-1);
    DEC(delay,MDP.delay-1);MDP.delay:=0;
  ELSE MDP.delay:=0
  END;
  INC(wait,delay);delay:=0;
END IFUDelay;

PROCEDURE BSIR8;
BEGIN
  IFUDelay;
  IF boot THEN bus:=CARDINAL(bootROM[pc MOD 2048]); INC(pc) ELSE
  bus:=CARDINAL(reg.bytes[(pc+(flo MOD 4) * 2) MOD 8]);
  INC(pc);
  IF ((pc + (flo MOD 4) * 2) MOD 8) = 0 THEN
    Memory.Read64(fhi,(pc DIV 2 + flo),reg.words);INC(memcnt);
    delay:=Memory.cycle + MDP.delay;idle:=0;
    IF MDP.delay >0 THEN DEC(delay) END;
    (* if overflow, offset has crossed page boundary*)
  END;
 END;
 INC(ifubytes);
END BSIR8;

PROCEDURE BSIRMINUS;
BEGIN
  BSIR8;
  bus:=bus + 0FF00H
END BSIRMINUS;

PROCEDURE BSIR4;
BEGIN
  ALU.qloaded:=TRUE;
  INC(mcodedelay,delay);
  IF idle<2 THEN INC(delay1,idle) ELSE INC(delay1,2) END;
  IFUDelay;
  IF boot THEN bus:=CARDINAL(bootROM[pc MOD 2048]); INC(pc) ELSE
  bus:=CARDINAL(reg.bytes[(pc+(flo MOD 4) * 2) MOD 8]);
  INC(pc);
  IF ((pc+(flo MOD 4) * 2) MOD 8) = 0 THEN
    Memory.Read64(fhi,(pc DIV 2 + flo),reg.words);INC(memcnt);
    delay:=Memory.cycle +1+ MDP.delay; idle:=0;
    IF MDP.delay >0 THEN DEC(delay) END;
    (* if overflow, offset has crossed page boundary*)
  END;
 END;
 INC(ifubytes);
 INC(mcodecnt);
END BSIR4;

PROCEDURE BSIRSTAR;
BEGIN
  IFUDelay;
  bus:=CARDINAL(reg.bytes[(pc+(flo MOD 4) * 2) MOD 8]);
  INC(pc);INC(ifubytes);
END BSIRSTAR;

PROCEDURE LoadPC;
VAR i:CARDINAL;
BEGIN
  IFUDelay;
  pc:=bus;
  boot:=0 IN BITSET(pc);
  IF boot THEN pc:=pc MOD 32768 END;
  Memory.Read64(fhi ,(pc DIV 2 + flo),reg.words);
  INC(memcnt);delay:=Memory.cycle + MDP.delay;idle:=0;
  IF MDP.delay >0 THEN DEC(delay) END;
END LoadPC;

PROCEDURE LoadF;
BEGIN
  IFUDelay;
  f:=bus;
  fhi:=f DIV 8000H;
  flo:= (f MOD 8000H)*2;
END LoadF;
```

```
PROCEDURE ReadFileAndStoreInROM(fileName:ARRAY OF CHAR);
VAR i,tmp,lineLength,sum,address,highAddressByte : CARDINAL;
     c : CHAR;
     f : File;

  PROCEDURE ReadByte():CARDINAL;
  VAR byte : [0..255];

  PROCEDURE ReadHex():CARDINAL;
  VAR c : CHAR;
    BEGIN
      c:=0C;
      WHILE (c<'0') DO ReadChar(f,c);END;
      IF c >='A' THEN
        RETURN (ORD(c)-ORD('A')+10);
      ELSE
        RETURN (ORD(c)-ORD('0'));
      END;
    END ReadHex;

  BEGIN
    byte:=ReadHex()*16;
    RETURN(CARDINAL(byte+ReadHex()));
  END ReadByte;

BEGIN
  Lookup(f,fileName,FALSE);
  IF f.res = notdone THEN
    WriteLn;WriteString(fileName);WriteString(' not found.');
    HALT;
  END;
  sum:=0; address:=0;
  LOOP
    REPEAT
      ReadChar(f,c);
    UNTIL c = 'S';
    ReadChar(f,c);
    IF c = '9' THEN EXIT; ELSIF c <> '1' THEN
      WriteString(' S1 missing at head of line');
      HALT;
    END;
    lineLength := ReadByte()-3;
    highAddressByte := ReadByte();
    tmp:=highAddressByte*256+ReadByte();
    IF address <> tmp THEN
      address:=tmp;
    END;
    FOR i:=1 TO lineLength DO
      tmp := ReadByte();
      bootROM[address]:= CHAR(tmp);
      INC(address);
    END; (* FOR *)
    ReadChar(f,c);ReadChar(f,c);ReadChar(f,c);
    IF c <> 36C THEN WriteString(' EOL missing '); HALT;END;
    sum:=0;
  END; (* LOOP *)
  Close(f);
END ReadFileAndStoreInROM;
```

```
BEGIN
  boot:=FALSE;
  pc:=0;
  f:=0;fhi:=0;flo:=0;ifubytes:=0;wait:=0;memcnt:=0;delay:=0;idle:=0;
  mcodecnt:=0;delay1:=0;mcodedelay:=0;waitcnt:=0;
  ReadFileAndStoreInROM('DK.int13dot2.BOT');
END IFU;

MODULE MDP;
  IMPORT bus,Memory,IFU;
  EXPORT QUALIFIED ReadMDPData,LoadMDPWriteData,LoadHMAR,LoadMAR,hmar,
         delay,wait;
  VAR mdpData,mdpWriteData,hmar,mar,wait,delay : CARDINAL;
      writeFlag:BOOLEAN;

  PROCEDURE MDPDelay;
  BEGIN
    IF IFU.delay > delay
    THEN DEC(IFU.delay,delay)
    ELSE INC(IFU.wait,IFU.delay);
      DEC(delay,IFU.delay);IFU.delay:=0
    END;
    INC(wait,delay);
    delay:=0;
  END MDPDelay;

    PROCEDURE LoadMDPWriteData;
  BEGIN
    MDPDelay;
    mdpWriteData:=bus;
    writeFlag:=TRUE;
  END LoadMDPWriteData;

  PROCEDURE LoadHMAR;
  BEGIN
    hmar:=bus;
  END LoadHMAR;

  PROCEDURE LoadMAR;
  BEGIN
    MDPDelay;
    IF writeFlag THEN
      Memory.Write(hmar,bus,mdpWriteData);
      writeFlag:=FALSE
    ELSE
      mdpData:= Memory.Read(hmar,bus)
    END;
    hmar:=0;
    delay:=Memory.cycle+IFU.delay+ 1;
  END LoadMAR;

  PROCEDURE ReadMDPData;
  BEGIN
    MDPDelay;
    bus:=mdpData;
  END ReadMDPData;

BEGIN
```

```
  hmar:=0;writeFlag:=FALSE;delay:=0;wait:=0;
END MDP;

MODULE Memory;
FROM  SYSTEM     IMPORT WORD;
FROM  WinDisk   IMPORT DiskRead,DiskWrite;
FROM  FileSystem IMPORT File,Lookup,ReadWord,Response,Close;
FROM Terminal IMPORT WriteString;
EXPORT QUALIFIED Read,Read64,Write,readcnt,writecnt,cycle;

CONST maxPages        =    16; (*number of memory pages resident*)
      unloaded  = 1024;
      dumpFileOffset  =  192; (*starting sector of disk resident memory*)
      diskDrive   =    0;
      cycle = 7;

TYPE PageIndex  = [0..1024];
     (* 1024 is used as the index for unused pages.*)
     Page       = ARRAY [0..127] OF CARDINAL;
     (* Pages are the same size as disk sectors.   *)

VAR
    diskstatus           : Response;
    i,size,adr,help      : CARDINAL;
    oldestPage           : CARDINAL;
    writecnt ,readcnt    : CARDINAL;
    f                    : File;
    memory               : ARRAY [0..    1023] OF PageIndex;
    pageTable            : ARRAY [0..maxPages-1] OF RECORD
                                                   pagenum  : CARDINAL;
                                                   written: BOOLEAN;
                                                   page     : Page;
                                                 END;

PROCEDURE SXFW(h,l,d:CARDINAL);
CODE 223B;
END SXFW;

PROCEDURE Read( hmar,mar:CARDINAL):CARDINAL;
VAR srcpage,d:CARDINAL;
BEGIN
  INC(readcnt);
  srcpage := (hmar*512)+(mar DIV 128);
  IF memory[srcpage]=unloaded THEN
    SwapPage(srcpage);
  END;
  RETURN(pageTable[memory[srcpage]].page[mar MOD 128]);
END Read;

PROCEDURE Read64( hmar,mar:CARDINAL;VAR blk:ARRAY OF WORD);
VAR srcpage:CARDINAL;
BEGIN
  INC(readcnt);
  srcpage := (hmar*512)+(mar DIV 128);
  mar:=CARDINAL(BITSET(mar) * {0..13});
  IF memory[srcpage]=unloaded THEN
    SwapPage(srcpage);
  END;
  FOR i:=0 TO 3 DO
```

```
    blk[i]:= WORD(pageTable[memory[srcpage]].page[mar MOD 128 + i])
  END;
END Read64;

PROCEDURE Write(hmar,mar,data:CARDINAL);
VAR dstpage:CARDINAL;
BEGIN
  INC(writecnt);
  dstpage := (hmar*512)+(mar DIV 128);
  IF hmar>0 THEN SXFW((hmar MOD 2)*4000H,mar,data) END;
  IF memory[dstpage]=unloaded THEN
    SwapPage(dstpage)
  END;
  pageTable[memory[dstpage]].page[mar MOD 128]:=data;
  pageTable[memory[dstpage]].written:=TRUE;
END Write;

PROCEDURE SwapPage(num:CARDINAL);
BEGIN
  IF pageTable[oldestPage].written THEN
    DiskWrite(diskDrive,pageTable[oldestPage].pagenum+dumpFileOffset,
              pageTable[oldestPage].page,diskstatus)
  END;
  memory[pageTable[oldestPage].pagenum]:=unloaded;
  LoadPage(num);
END SwapPage;

PROCEDURE LoadPage(num:CARDINAL);
BEGIN
  memory[num]:=oldestPage;
  oldestPage:=(oldestPage + 1) MOD maxPages;
  pageTable[memory[num]].pagenum:=num;
  pageTable[memory[num]].written:=FALSE;
  DiskRead(diskDrive,dumpFileOffset + num,
     pageTable[memory[num]].page,diskstatus);
END LoadPage;

BEGIN
  FOR i:=0 TO 1023 DO memory[i]:=unloaded END;
  oldestPage:=0;
  FOR i:= 0 TO maxPages-1 DO LoadPage(i) END;
  oldestPage:=2;
  Lookup(f,'DK.simtest.ABS',FALSE);
  IF f.res <> done THEN WriteString('no boot file') END;
  ReadWord(f,size);
  WHILE size <> 0 DO
    ReadWord(f,adr);
    FOR i:= 2 TO size DO
      ReadWord(f,help);
      Write(0,adr,help);
      INC(adr) END;
    ReadWord(f,size);
  END;
  readcnt:=0;
  writecnt:=0;
END Memory;

PROCEDURE ExecuteCycles(stopCondition:INTEGER);
```

```
  PROCEDURE Stop():BOOLEAN;
  BEGIN
    IF stopCondition = 0 THEN RETURN TRUE
    ELSIF stopCondition <0 THEN
      RETURN (MIR.bs = bsIR4) OR (MIR.bs = bsINT)
    ELSE DEC(stopCondition); RETURN FALSE
    END
  END Stop;

BEGIN
  REPEAT
    IF MIR.k THEN ALU.ClockALU;
      CASE MIR.bd OF
    bdDASH: |
    bdALU :  |
    bdSR  : ALU.shiftCountReg:=bus MOD 16|
    bdPC  : IFU.LoadPC|
    bdF   : IFU.LoadF|
    bdINM : iMask:=bus MOD 256 |
    bdIOA : ioAddress := bus MOD 64;|
    bdIOD : IO.Put |
    bdBNK : MDP.hmar := bus MOD 2|
    bdMAR : MDP.LoadMAR|
    bdMD  : MDP.LoadMDPWriteData |
    bdMDS :
      ELSE
        WriteLn;WriteString(' sim: Illegal value in dest ');
        HALT;
      END; (* CASE *)
    END;
    IF MIR.sp THEN INC(time,3);
      IF IFU.delay>0 THEN DEC(IFU.delay) ELSE INC(IFU.idle) END;
      IF MDP.delay>0 THEN DEC(MDP.delay) END;
    ELSE INC(time,2);
    END;
    IF IFU.delay>0 THEN DEC(IFU.delay) ELSE INC(IFU.idle) END;
    IF MDP.delay>0 THEN DEC(MDP.delay) END;
    IF IFU.delay>0 THEN DEC(IFU.delay) ELSE INC(IFU.idle) END;
    IF MDP.delay>0 THEN DEC(MDP.delay) END;
    IF time>65000 THEN WriteState END;
    MIR.LoadMIR;
    bus:=0FFFFH;

    IF MIR.k THEN
      IF MIR.e THEN bus:=MIR.n;MIR.bd:=bdALU;MIR.bs:=bsDASH;
         INC(ecnt) END;
      CASE MIR.bs OF
    bsALU     : IF MIR.bd = bdALU THEN
                   IF MIR.dest = ALU.RAMA THEN bus:=ALU.Reg[MIR.a]
                   ELSE WriteString('bad dest value');HALT END;
                 ELSE ALU.EvaluateALU; bus:= ALU.y ;
                   ALU.qcount:=ALU.lastqcnt
                 END|
    bsMD      : MDP.ReadMDPData |
    bsIR4     : IF interrupt THEN MIR.bs:=bsINT
               ELSE IFU.BSIR4 END;
               IF ucnt >19 THEN INC(mcnt[20]) ELSE INC(mcnt[ucnt]) END;
```

```
                ucnt:=0 |
    bsIR8     : IFU.BSIR8 |
    bsIRSTAR  : IFU.BSIRSTAR |
    bsIRMINUS : IFU.BSIRMINUS |
    bsF       : bus := IFU.f|
    bsPC      : bus := IFU.pc|
    bsINM     : bus := iMask|
    bsIOD     : IO.Get |
    bsDASH    :
      ELSE
        WriteString('Invalid value in bs');HALT
      END; (* CASE *)
      ALU.EvaluateALU;
    END;
    INC(ucnt);
    MCU.NextUAD;
  UNTIL Stop();
END ExecuteCycles;

PROCEDURE Reset;
  VAR i:CARDINAL;
  BEGIN
    interrupt:=FALSE; uad:=1;time:=0;;
    Memory.Write(0,14B,0C000H);
    Memory.Write(0,15B,0);
    i:=106100B;
    REPEAT
      Memory.Write(0,i,0);INC(i)
    UNTIL i= 106100B + 1024 ;
    MIR.LoadMIR;
    bus:=0FFFFH;
    IF MIR.k THEN
      IF MIR.e THEN bus:=MIR.n;MIR.bd:=bdALU;MIR.bs:=bsDASH END;
      CASE MIR.bs OF
    bsALU     : IF MIR.bd = bdALU THEN
                   IF MIR.dest = ALU.RAMA THEN bus:=ALU.Reg[MIR.a]
                   ELSE WriteString('bad dest value');HALT END;
                 ELSE ALU.EvaluateALU; bus:= ALU.y END|
    bsMD      : MDP.ReadMDPData |
    bsIR4     : IF interrupt THEN MIR.bs:=bsINT
                ELSE IFU.BSIR4 END |
    bsIR8     : IFU.BSIR8 |
    bsIRSTAR  : IFU.BSIRSTAR |
    bsIRMINUS : IFU.BSIRMINUS |
    bsF       : bus := IFU.f|
    bsPC      : bus := IFU.pc|
    bsINM     : bus := iMask|
    bsIOD     : IO.Get |
    bsDASH    :
      ELSE
        WriteString('Invalid value in bs');HALT
      END; (* CASE *)
      ALU.EvaluateALU;
    END;

    MCU.NextUAD;
END Reset;
```

```
BEGIN
  Reset;
  ecnt:=0;
  FOR i:=1 TO 20 DO mcnt[i]:=0 END;
  ucnt:=1;
  LOOP
    ExecuteCycles(oneCode);
  END;
END Simulator.
```

Appendix 3 A Performance Measurement Program

This program first appeared in the publication, "The Personal Computer Lilith", by N. Wirth [Wir2] where it was used to measure the performance of Lilith in comparison with several other computers. The program was subsequently used to measure other computers and the results were published in Jacobi's dissertation [Jac]. The program as it has been modified here has been used to develop new numbers for Lilith with the 375 nanosecond main memory, and it has been used by the simulation program in Appendix 2 as the basis for the analysis of Lilith's dynamic performance. The results of this analysis were reported in Chapter 5.

Some additional tests results have been provided by Roger Vossler of TRW and Dave Coar of Floating Point Systems, using machines at their installations. Vossler provided the measurements of the VAX 750, the PDP11/45, and the SUN computers. Coar provide measurements of a 10 megahertz 68000 computer.

Test	Lilith	PDP11/45	VAX750	SUN	VAX750	68000
	375 ns	C	C		C	PASCAL
MODULA-2						
a empty REPEAT loop	464	314	433	483	463	502
b empty WHILE loop	480	210	468	338	497	501
c empty FOR loop	652	251	422	376	363	578
d CARDINAL arithmetic	226	118	153	40	160	162
e REAL arithmetic	147	138	100	9	84	
f sin,exp,ln,sqrt	103	37	26	3	51	
g array access	161	77	137	93	55	125
h with bound test	126					102
i matrix access	269	140	120	59	112	210
j with bound test	214					175
k call of empty proc.	214	63	101	293	64	233
l with 4 parameters	144	42	79	133	56	142
m copying arrays	113				85	96
n access via pointer	181				215	
o reading disk stream	117					

The 10 megahertz 68000 based system is most nearly equal to Lilith in performance, with the exception of its real arithmetic. Real arithmetic on the 68000 must be implemented in software and therefore is extremely slow as the numbers from the SUN 68000 demonstrate. The comparison between these machines demonstrates most dramatically the advantage Lilith has from its high level language oriented architecture. The 10 mhz 68000 has a microinstruction cycle time of 100 nanoseconds, which is 50% faster than Lilith, and it has a memory cycle time of 400 nanoseconds compared to the Lilith's memory cycle time of 375 nanoseconds. Even with these advantages, it still is only almost as fast as Lilith.

```
MODULE Benchmark;
 (* modified for use with SIMULATOR program, R. Ohran, Nov,1983 *)
 (*$T-
   a: empty REPEAT loop
   b: empty WHILE loop
   c: empty FOR loop
   d: CARDINAL arithmetic
   e: REAL arithmetic
   f: standard functions
   g: array of single dimension
   h: same as g but with index tests
   i: matrix access
   j: same as i but with index tests
   k: call of empty, parameterless procedure
   l: call of empty procedure with 4 parameters
   m: copying of arrays ( blockmoves)
   n: pointer chaining
   o: reading of file  *)
FROM SYSTEM IMPORT WORD;
FROM Storage IMPORT ALLOCATE;
FROM Terminal IMPORT Read,BusyRead,Write,WriteLn;
FROM InOut IMPORT WriteCard;
FROM FileSystem IMPORT File,Lookup,ReadWord,Reset,Response;
FROM MathLib0 IMPORT sin,exp,ln,sqrt;
FROM DisplayDriver IMPORT Show,BMD;

TYPE NodePtr = POINTER TO Node;
     Node = RECORD x,y:CARDINAL; next: NodePtr END;

VAR A,B,C: ARRAY[0..255] OF CARDINAL;
    M: ARRAY [0..99],[0..99] OF CARDINAL;
    m,i: CARDINAL;
    head: NodePtr;

  PROCEDURE GET(chan : CARDINAL; VAR value : WORD);
  (*---------*)
  CODE 240b
  END GET;

  PROCEDURE PUT(chan : CARDINAL; value : WORD);
  (*----------*)
  CODE 241b
  END PUT;

PROCEDURE Test(ch:CHAR);
  VAR i,j,k: CARDINAL;
      r0,r1,r2: REAL; p: NodePtr;

  PROCEDURE P;
    BEGIN
    END P;

  PROCEDURE Q(x,y,z,w: CARDINAL);
    BEGIN
    END Q;

BEGIN
  CASE ch OF
```

```
"a": k:=20000;
     REPEAT
       k:=k-1
     UNTIL k = 0 |

"b": i:= 20000;
     WHILE i > 0 DO
       i:= i-1
     END  |

"c": FOR i:= 1 TO 20000 DO
     END |

"d": j:= 0; k:= 10000;
     REPEAT
       k:=k-1; j:=j+1; i:=(k*3) DIV (j*5)
     UNTIL k = 0 |

"e": k:=5000; r1 := 7.28; r2 := 34.8;
     REPEAT
       k:=k-1; r0 := (r1*r1) / (r1+r2)
     UNTIL k = 0  |

"f": k:= 500;
     REPEAT
       r0 := sin(0.7); r1:= exp(2.0);
       r0 := ln(10.0); r1 := sqrt(18.0); k:= k-1
     UNTIL k = 0 |

"g": k:=20000;i:=0 ; B[0] :=73;
     REPEAT
       A[i]:=B[i];B[i]:=A[i];k:= k-1
     UNTIL k = 0 |

"h": (*$T+*) k:=20000;i:=0 ; B[0] :=73;
     REPEAT
       A[i]:=B[i];B[i]:=A[i];k:= k-1
     UNTIL k = 0  (*$T-*)|

"i": FOR i:= 0 TO 99 DO
       FOR j:= 0 TO 99 DO
         M[i,j] := M[j,i]
       END
     END  |

"j": (*$T+*)FOR i:= 0 TO 99 DO
       FOR j:= 0 TO 99 DO
         M[i,j] := M[j,i]
       END
     END (*$T-*) |

 "k": k:=20000;
     REPEAT
       P;k:=k-1;
     UNTIL k=0  |

"l": k:=20000;
     REPEAT
       Q(i,j,k,m); k:=k-1;
     UNTIL k=0  |
     "m": k:=500;
          REPEAT
            k:=k-1; A:=B; B:=C; C:=A;
          UNTIL k =0 |

     "n": k:=500;
          REPEAT p:=head;
            REPEAT p:=p↑.next UNTIL p= NIL;
            k:=k-1;
          UNTIL k = 0 |

    "o": k := 5000;
          REPEAT
            k:=k-1; ReadWord(f,i)
          UNTIL k = 0;
          Reset(f)
  ELSE
  END
END Test;

CONST display = 74B;
      off = FALSE;
      on  = TRUE;

VAR ch,ch1: CHAR;
    n: CARDINAL;
    f: File;
    q: NodePtr;

BEGIN
  Lookup(f,"anyFile",FALSE);
  head:=NIL; n:=100;
  REPEAT q:=head;
    NEW(head); head↑.next:=q; n:=n-1
  UNTIL n=0;
  Write('>'); Read(ch);
  WHILE ("a" <= ch) & (ch <= "p") DO
    Write(ch); WriteLn; n:=0;
    PUT(display,on);
    IF ch<'p' THEN
      REPEAT n:=n+1; Test(ch);
(*          IF(n MOD 50) = 0 THEN WriteLn END;
        Write('.');*)BusyRead(ch1);
      UNTIL ch1 <> 0C;
    ELSE
      Write('p');
      FOR i:=1 TO 10 DO
        Test('a');Test('b');Test('c');Test('d');
        Test('e');Test('g');Test('h');Test('i');
        Test('j');Test('k');Test('l');Test('m');
      END;
    END;
    PUT(display,on);
    WriteCard(n,6);WriteLn; Write(">"); Read(ch)
  END ;
END Benchmark.
```

References

[Am] Bipolar Microprocessor Logic and Interface Data Book, Advanced Micro Devices 1982.

[Amm] U. Ammann, Die Entwicklung eines Pascal-Compilers nach der Methode des Strukturierten Programmierens, Diss ETH 5456, ETH Zurich, 1975.

[BS] Miles A. Barel, John P. Strait, PERQ QCode Reference Manual, Three Rivers Computer Corporation, Sept 3, 1980.

[Bar] R. S. Barton, "A New Approach to the Functional Design of a Digital Computer", Proceedings of the Western Joint Computer Conference, 1961, pp. 393-396.

[Bul] D. M. Bulman, "Stack Computers:An Introduction", Computer, IEEE Computer Society, Long Beach, California. May 1977, pp.18-28.

[Bur] Burroughs Corp., Burroughs B5500 Reference Manual, Detroit, 1968.

[Bur2] The Operational Characteristics of the Processors for the Burroughs B 5000, Burroughs Corporation, Nov. 1961.

[Dij] E. W. Dijkstra, Recursive Programming, Numerische Mathematik 2, 1960, pp. 312-318.

[For] Forest Baskett, Puzzle: an informal compute bound benchmark, Widely used and circulated.

[Gei] L. Geissmann, Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith, Diss. 7286, ETH, Zurich,1983.

[Ham] C. L. Hamblin, "An Addressless Coding Scheme Base on Mathematical Notations," Proc., W. R. E. Conference on Computing, Salisbury, South Australia, June 1957.

[Han82] L. Geissmann, J. Hoppe, C. Jacobi, S. E. Knudsen, W. Winiger, N. Wirth, Lilith Handbook, Institut für Informatik, ETH, Zurich, October 1982.

[Hop83] J. Hoppe, Magnet: A Local Network for Lilith Computers, Institut für Informatik, ETH, Zurich, 1983.

[Hop2] J. Hoppe, Maintenance and Testing of Lilith, Report 58, Institut für Informatik, ETH, Zurich, December,1983.

[HP] Hewlett-Packard Co., HP3000 Computer System Reference Manual, Cupertino, 1973.

[Int] Intel Component Data Catalog, 1982, pp. 5-8.

[Int2] Reference book on Intel 8008.

[Jac] Christian Jacobi, Code Generation and the Lilith Architecture. Diss ETH No 7195. Swiss Federal Institute of Technology, Zurich.1982, p. 1.

[Jac2] Christian Jacobi, Evaluation of the Lilith Architecture in View of Compiling Modula Programs, Proceedings of the German Chapter of the ACM 11, Teubner, July 1982.

[Le] Van Kiet Le, The Module: A Tool for Structured Programming, Diss ETH No.6153, ETH, Zurich, 1978.

[Knu] S. E. Knudsen, Medos-2: A Modula-2 Oriented Operating System for the Personal Computer Lilith, Diss ETH No. 7346, ETH, Zurich, 1983.

[Mick] John Mick and Jim Brick. Bit-Slice Microprocessor Design. McGraw-Hill, Inc, 1980, pp. 93-127.

[NAJNJ] K. V. Nori, U. Ammann, K. Jensen, H. H. Naegeli, Ch. Jacobi, The Pascal-P Implementation Notes, Pascal - The Language and Its Implementation, Edited by D. W. Barron, John Wiley

[Ohr] Richard Ohran, Lilith Hardware Manual, Modula Research Institute, Provo, Utah, 1982.

[Ohr2] Richard Ohran, Lilith and Modula-2, Byte Magazine, August 1984, p. 181-192.

[TI] Texas Instruments, The Bipolar Microcomputer Components Data Book, Second Edition, 1979.

[TMLSB] C. P Thacker, E. M. McCreight, B. W. Lampson, R. R. Sproull, and D. R. Boggs, Alto: A personal computer, Xerox, Palo Alto, California. July 1979, p. 14.

[Wir71] N. Wirth, The Programing Language Pascal, Acta Informatica 1, 1971, pp. 35-63.

[Wir1] N. Wirth, Modula: A Language for Modular Multiprogramming, Software-Practice and Experience, Jan 1977, pp. 3-35.

[Wir2] N. Wirth, The Personal Computer Lilith, Report 40, Swiss Federal Institute of Technology, Zurich. 1981.

[Wir82] N. Wirth, Programming in Modula-2, Springer-Verlag, Berlin Heidelberg, 1982.

Curriculum Vitae

Richard Stanley Ohran
125 West 4750 North
Provo, Utah 84604

I was born on June 11, 1942 in the city of San Mateo, California, USA. From 1948 until 1960 I attend primary and secondary school in Belmont, California. I began college in 1960 at Brigham Young University. My college career was interrupted for a period of 30 months in 1963 during which time I performed voluntary service for my church. In 1968 I graduated from Brigham Young University with a five year bachelor's engineering degree in electronics.

During the 1968-1969 school year I attended graduate school at the University of Illinois.

Since 1970 until now I have held a teaching position at Brigham Young University in Provo, Utah. From 1970 until 1981 my position was with the Electrical Engineering Department in the capacity of an instructor. Since 1981 I have been an assistant professor in the Computer Science Department. All other employments which I have have since 1970 have been in addition to my position as a university faculty member.

From 1974 until 1975 I was employed part time as a visiting instructor at the Computer Science Department of the University of Utah. While working at the University of Utah, I was also enrolled in the Master's program and completed a Masters degree in computer science in 1977.

From 1975 until 1978 I was employed during the summers as a Unesco field expert in computers at the Polytechnic of Bucharest in Romania.

From the fall of 1977 until January, 1981 I spent a sabbatical leave at the Institut für Informatik of the Swiss Federal Institute of Technology(ETH) in Zurich.

Since September, 1981 I have been the director of the Modula Research Institute alongside of my teaching responsibilities at BYU.

During the last half of 1983 and the first part of 1984, I have been on leave from BYU and employed at the Modula Corporation.

Since December 1966, I have been married to Maralee Young and have a family of six sons.