UNIVERSITY OF ILLINOIS
DIGITAL COMPUTER LABORATORY
URBANA, ILLINOIS

ORDER CODE FOR THE NEW ILLINOIS COMPUTER

by

D. B. Gillies

File No. 458
June 15, 1962

TABLE OF CONTENTS

Page

1. Introduction 1

Present Status 1
Completion Schedule (tentative). 2
Preliminary Remarks on Programming 2

2. General Mode of Operation. 3

Delayed Control. 3
Advanced Control 5
Interplay. 5
Program Interrupt 9

3. Order Code for Floating Point Arithmetic 10

The Floating Point Accumulator 12
Zero and Overflow 13
Normalization. 15
Addition and Subtraction 15
Multiplication 16
Division 16
Round-Off. 17
Correct Overflow and Detect Zero 18

4. Orders Which Do Not Involve Floating Point 24

Modifier Arithmetic. 31

APPENDIX 33

1. Introduction

This computer, which is now nearing completion, will have the following general characteristics:

Word Length 52 bits

Arithmetic Floating point (multiply time 5 to 10 microseconds)

Instruction length 13 or 26 bits

Address length 13 bits

Index registers (also called Modifiers) 16, each 13 bits long

Main Memory 8192 (or 2^{13}) words of core memory arranged in two, 4096 word memories.

Memory Cycle 1.8 μsec for each memory

Fast Memory 10 words, 0.2 μsec access time

Back-up Memory Two drums 65,536 words total, 7 μsec/word
Two disc files about 10 million words total
10 IBM 729 MK VI magnetic tape units

Input	paper tape	1,000 alphanumeric characters/sec
	punch card	800 cards/minute
Output	line printer	600 lines/minute
	punch card	250 cards/minute
	paper tape	110 characters/sec

Mode of Operation Parallel, highly concurrent

Special Features Interrupt, memory protection, I/O protection

Possible Later Additions I/O from remote stations, oscilloscope I/O direct link to IBM 7094 computer and to pattern recognition computer now being designed at the Digital Computer Laboratory, larger core memory, faster arithmetic.

Present Status

One, 4096 word core memory is built and running at a cycle time between 1.6 and 2.0 μsec (microseconds).

Paper tape I/O is built and running.

Fast Memory is built and debugged.

Present Status (continued)

Floating point arithmetic unit is built and nearly all instructions are debugged. Present multiply time is about 12 μsec.

Floating point arithmetic is performed by the arithmetic unit under the control of "Delayed Control." A simple overall control called ACO executes just one instruction per word, without indexing, is now sequencing the above equipment.

Completion Schedule (tentative)

Summer 1962	Advanced Control, which incorporates the final order code, and has the ability to operate memory in parallel with arithmetic.
Fall 1962	Interplay, which allows up to 32 concurrently operating I/O channels or back-up memory channels to time-share the core memories. Full interrupt system. Speed-up of arithmetic.
Winter 1962-1963	First 32,768 words of magnetic drum memory, core memory protection equipment.
Spring 1963	Line printer, card reader and punch, and 4 magnetic tape units.
Summer 1963	Second core memory, second drum, 6 more magnetic tape units.
Fall 1963	Disc files.

Preliminary Remarks on Programming

From the above specifications, this is a very high speed scientific computer with a core memory which is small for machines in its class, and with a powerful system of back-up memories. It is hoped that the disc files can be modified to each deliver a word every 20 μsec.

In the past decade, the speed of computer arithmetic has gone up by a factor of 100, memory speed has gone up by a factor of 10, memory size by a factor of 10, and the speed of input-output has increased by a still smaller factor. The resulting machine imbalance can be offset partly through hardware by means of

(a) provision of many concurrently operating devices, together with interrupt and protection equipment

(b) fast temporary storage registers and a compact order code to reduce the number of memory references,

(c) provision of adequate storage space so data to or from busy devices may be stacked up for future use,

and partly through programming by such means as

(d) foreseeing future data requirements and calling for transfers between core memory and peripheral equipment early enough that the data is always in core memory when needed

(e) formulating problems in such a way as to reduce references to backing memories

(f) switching to another problem either because it has a higher priority or because the present problem would leave the machine idle (Multiprogramming).

Since a fast computer is also an economical computer, there is the problem of making the machine more accessable to people, particularly people whose problems are not so great as to tax the entire computer system. Some worthwhile goals are

(a) Input routines and compilers which are efficient and easy to use.

(b) Short turn around time (possibly as low as 10 minutes) for short problems or automatic code checks from, say, 8 a.m. to midnight.

(c) Input-output from remote stations.

2. General Mode of Operation

The principal controls and data paths are shown in Figure 1. There are three main control units in this computer called Delayed Control, Advanced Control, and Interplay.

Delayed Control

Floating point arithmetic is performed in a double precision accumulator in the arithmetic unit under the control of Delayed Control. IN and OUT are word registers which contain respectively the operand for the next Delayed Control instruction, and the result of the last Delayed Control store order.

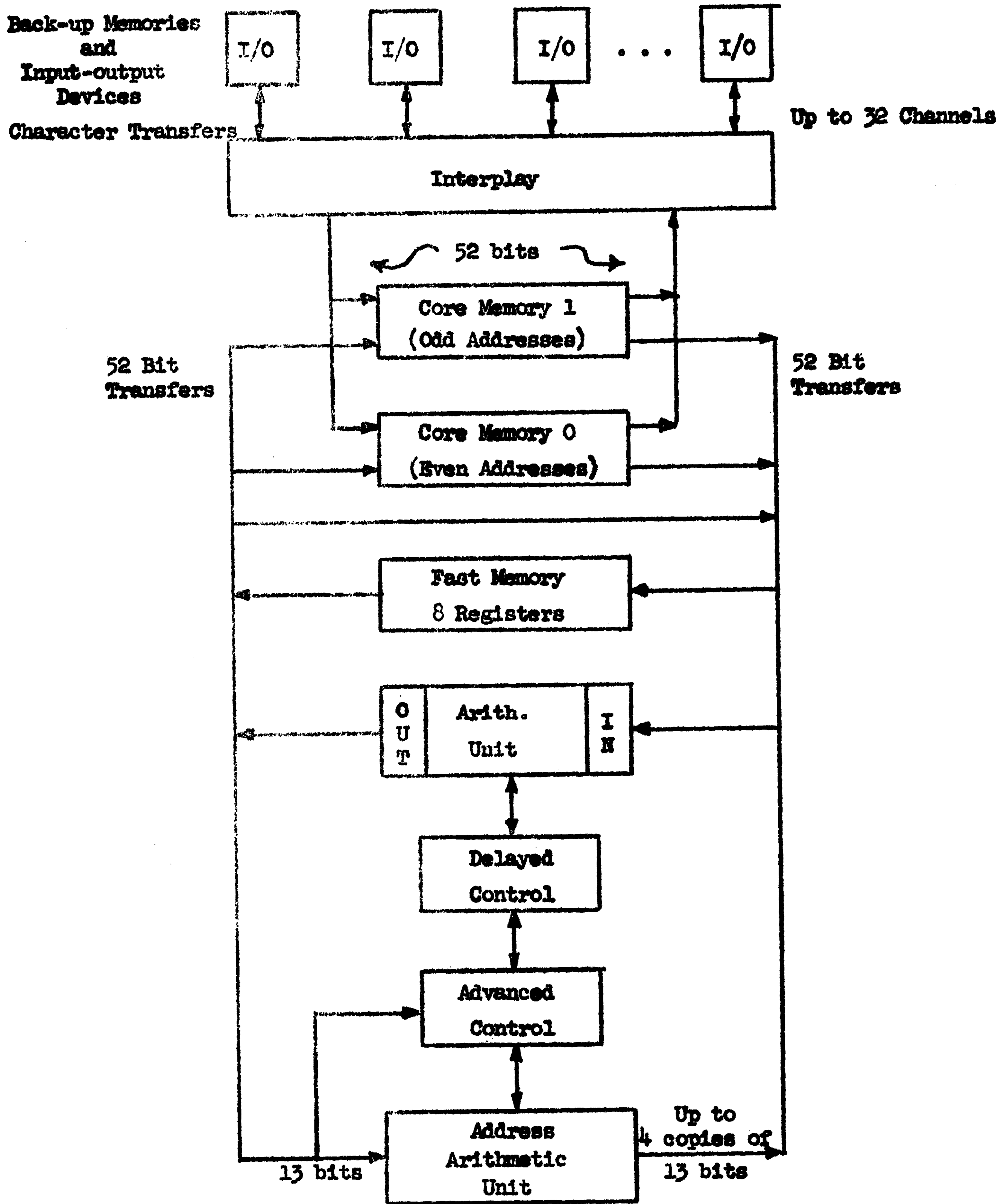

Figure 1

Data Paths for ILLIAC II

Advanced Control

Every instruction is obeyed first by Advanced Control. Some instructions, such as those which only change the value of a modifier register (i.e. index register) are obeyed in their entirety by Advanced Control using the [illegible]-bit wide address arithmetic unit. For orders obeyed by Delayed Control Advanced Control must form any address required, obtain any operand required and place it in the IN register in advance of the instruction execution by Delayed Control, and store any result from OUT after the instruction has been obeyed by Delayed Control. Advanced Control must also do the following:

(a) Transfer words of instructions from core memories into 2 registers called F8 and F9 in the fast memory.

(b) Sequence the control counter to define the core address and position inside the word of the present instruction.

(c) Prepare instructions destined for Interplay.

(d) Time-share the core memories with Interplay.

(e) Program interrupt, to be explained later.

Interplay

The basic Interplay operation is a block transfer between the core memory and any one of the input/output devices or back-up memories. This operation requires one interplay channel, which contains counters, word assembly equipment, and provision for accessing core memory and sensing the end of a block transfer. Most devices have their own private Interplay channel. In the case of magnetic tape units, there will be several units associated with a channel, at most one connected to the channel at any one time. Any number of channels may be running simultaneously, and in this case the core memories are time-shared among the various Interplay channels and Advanced Control. For an Interplay order, Advanced Control constructs an address in the address arithmetic unit (AAU) and sends it on to Interplay.

NOTE. In the case of the paper tape reader and punch, there is also a direct path to or from the AAU, so Advanced Control may also perform single tape character reads or punches independent of Interplay.

Core Memory #0. The memory containing all even-numbered locations: 0, 2, 4, ..., 8190.

Core Memory #1. The memory containing all odd-numbered locations: 1, 3, 5, ..., 8191.

> Note: For the period when only 1 core memory is attached to the machine, its locations are numbered 0, 1, 2, ..., 4095, and higher addresses refer to locations in this memory modulo 4096 so address 4096 refers to location 0, 4097 refers to 1, ..., 8191 refers to 4095.

Fast Memory. Registers called F0, F1, ..., F9 are specialized in purpose. F8, F9 contain words of instructions currently being obeyed by the computer. F8 holds the contents of some even-numbered location from Core Memory #0, and F9 holds the contents of the next higher numbered memory location: its address is odd so it came from Core Memory #1. Each of these registers is subdivided into 4, 13-bit fields called control groups. An instruction is made up of one or two control groups, and is classified as short or long respectively. Reading from left (most significant) to right in a word, the control groups are numbered 0, 1, 2, 3. A long instruction occupies any two consecutive control groups in memory, without restriction. Thus word 2000, control groups 0 and 1 could hold a long instruction which would be obeyed from F8,0 and F8,1. Likewise 2000,3 and 2001,0 could hold a long instruction executed from F8,3 and F9,0 and locations 2001,3 and 2002,0 could hold a long instruction executed from F9,3 and (after automatic refill of F8 and F9) from F8,0. With the exception of the instructions CJF, CJS, to be explained later, the programmer cannot refer explicitly to F8 and F9. Their use is automatic.

> Note: For the period when only 1 core memory is attached to the machine, instructions are obeyed from F9.

F4, F5, F6, F7 are four registers which may sometimes be considered as full word registers or, more commonly, each is divided into 4, 13-bit fields called modifiers. These modifiers are numbered M0, M1, ..., M15 and, reading from left to right

F4 comprises M0, M1, M2, M3
F5 comprises M4, M5, M6, M7
F6 comprises M8, M9, M10, M11
F7 comprises M12, M13, M14, M15.

So, for example, the instruction CAM, M7, 15 (clear add modifier = CAM) would replace the rightmost 13 bits of F5 by the integer 15, and the instruction STR F6 would normalize, round-off and store the contents of the floating point accumulator into register F6, thereby overwriting modifiers M8, M9, M10, M11. In the latter case M8 will have most significant digit equal to 1 if the accumulator is negative, equal to zero if the accumulator is positive, and would be composed of all zeros if and only if the accumulator held zero.

F2, F3 are temporary storage registers used for constants or intermediate floating point results.

F1 (also called IN) and F0 (also called OUT) are closely associated with the floating point arithmetic unit, in the following way: Advanced Control pre-processes every instruction obeyed by the machine. Some, such as CAM above, it completely executes itself, using a 13-bit Address Arithmetic Unit for any arithmetic required. For instructions causing I/O actions, it constructs an address, and routes the modified instruction on to Interplay. For instructions involving the floating point arithmetic unit, Advanced Control constructs any address required, places any operand (obtained from core memory, or fast memory, or from the address itself) in the register F1, and places the order in a register called DCR.

Meanwhile, the floating point arithmetic unit, under Delayed Control, may be executing a previous instruction. When Delayed Control is ready to obey this instruction, it copies F1 into an internal register in the Arithmetic Unit, and decodes the order which is held in DCR. Now F1 still holds the operand read in so one can say in general that F1 contains the operand used by the last D.C. (Delayed Control) instruction. Therefore, for example, one could square the contents of memory location 200 with the program

CAD	200	(Clear accumulator, add (200))
MPY	F1	(Multiply by last operand).

Results of Delayed Control store orders are placed in F0 and subsequently copied by Advanced Control to their correct destinations. If the stated destination is F0, then no further copying is necessary. Thus F0 contains the last number stored from the Arithmetic Unit. Two further instructions LFR (Load Fast Register from core memory) and SFR (Store Fast Register into core memory) are needed to complete the description of what is legal and what is illegal in the use of F0 and F1.

(1) Delayed Control operands may come from any of F0 through F7 or from core memory.

(2) Delayed Control results may go to F0, F2, ..., F7 or core memory, but not F1.

(3) LFR can load F2, ..., F7 but not F0 or F1.

(4) SFR can store F0, F2, ..., F7 but not F1.

Interplay is responsible for reading and writing blocks of information between core memory and back-up memory or I/O devices, concurrent with arithmetic. For block transfer purposes, the memory may be considered divided into 32 blocks of 256 words each,

block 0 comprising locations 0 to 255
block 1 comprising locations 256 to 511

. .
. .
. .

block 31 comprising locations 7936 to 8191

Thus a full block begins at some multiple of 256 and ends just before the next multiple of 256. For transfers to or from drum or disc file, an entire block must be transferred at one time. For other devices an initial address not necessarily equal to a multiple of 256 may be used, and Interplay decides that the transfer is over when the next multiple of 256 is reached, or a stop indication is received from the device, whichever happens sooner. Initially, there will be one Interplay channel per device connected although it is also possible to have a number of devices on one channel. For input (or playback from drum or file) Interplay assembles characters into words, and periodically competes with Advanced Control for the use of memory to store one word. There is also a prior competition between the various interplay channels active at the time to see which will have the opportunity of competing for core memory. The completion of any block transfer causes Interplay to set an indicator which may cause program interrupt. (See below.)

Memory protection is accomplished by means of the Block Checker, a device having 32 indicators (called block flipflops), one for each 256-word block of memory. A block may be set busy (indicator on) because of an Interplay transfer in progress, or for any other reason. Subsequently all addresses going from

Advanced Control to the core memories are checked to be sure that a busy block is not being referred to. Reference to a locked out block when not in the interrupt mode (see below) is a program error and causes first read-out over write was requested, and second program interrupt.

Program Interrupt

Under certain conditions, some of which have already been described, it is desirable to break into a program and execute a different program retaining the option to resume the old program where we left off. The action of leaving the program and retaining such information as is required to resume it later is called program interrupt. Causes which might justify interrupt include the following:

(1) Correctable machine malfunctions, such as the incorrect read-in of a block from a magnetic tape unit. In this case it is very possible that a second reading of the same section of tape can be done error-free and it is convenient to have the system program handle this correction automatically for the programmer.

(2) The completion of a block transfer or tape rewind, etc. In this case the programmer or the system program may wish to give another block transfer to Interplay.

(3) Illegal order executed by Advanced Control. During program debugging it is desirable to print the location and contents immediately rather than allow control to proceed, perhaps obeying data as instructions. In a production run the occurrence of an illegal order means either a machine malfunction or that the program was not properly debugged.

(4) An unusual and possibly unwanted arithmetic result, such as floating point overflow.

(5) Periodic real-time signals furnished by a clock. This permits a system program to supervise code checks, and possibly keep a log, etc.

After interruption has taken place, the machine operates in a different mode called the interrupt mode, until a particular order is obeyed, (JDC with B=0 to be explained later). In this mode orders referring to Interplay and to the Block Checker are made legal, all references to busy blocks are legalized, and no

further interruptions may take place. An interrupt program determines the [illegible] of the interruption, takes appropriate steps to remedy the situation, and, if possible, resumes the program with a JDC, B=0 order which takes it out of the interrupt mode and back to the program.

A console switch, colored red, called the engineering switch, disables interrupt and makes everything legal except, of course, non-existent orders.

3. Order Code for Floating Point Arithmetic

A word consisting of instructions is divided into 4, 13-bit fields called control groups. An instruction consists of one or two control groups [illegible] is referred to as short or long respectively. A short instruction has 3 fields reading from left to right or most significant to least significant.

F 7 bits designating the operation to be performed. These bits may be designated by a 3-letter mnemonic such as MPY for multiply, or by one sexadecimal digit (base 16) followed by one octal digit. The paper tape code for the integers 0 - 15 is 0 ... 9 + - a b c d so MPY is +0 in sexadecimal, octal or 1010000 in binary.

B 4 bits usually designating a modifier register M_B or a fast register F_B.

C 2 bits which usually control address or operand preparation and may indicate whether the instruction is short or long.

A long instruction consists of F, B, C in one control group, and a second control group, called N which is usually an address.

Instructions destined for Delayed Control fall in 4 categories:

Full-word Arithmetic (such as MPY)

Full-word Store (such as STR: normalize, round and store)

Exponent Arithmetic and Shifts (such as ADE: add to exponent)

Quarter-word Store (the orders SIA: store integer part as an address; and SEX: store exponent).

Consider first the interpretation of B, C and possibly N for full-word arithmetic:

If $C = 0$, modifier M_B contains an address (M_B) of a core location containing the operand.

If $C = 1$, again (M_B) defines the core location containing the operand, but also $1 + (M_B)$ is returned to M_B.

If $C = 2$, a long instruction with core address $N + (M_B)$ mod 8192.

If $C = 3$ and $B < 8$ the operand is contained in fast register F_D.

If $C = 3$ and $B = 8$ the core address is N.

If $C = 3$ and $B = 9$ the integer N converted to floating point is itself the operand. In this case the leftmost digit of N is considered to have negative weight so $-4096 \leq N \leq 4095$.

If $C = 3$ and $B = 10$ the fraction N converted to floating point is the operand, and $-1 \leq$ operand $\leq 1 - 1/4096$.

If $C = 3$, $11 \leq B \leq 14$. Unassigned. At present has the effect that floating point zero is the operand, but these should not be used in programs because later additions to the computer might require the use of these combinations.

If $C = 3$, $B = 15$ floating point zero is the operand.

In summary, the instruction is short unless $C = 2$, or $C = 3$ and B is one of 8, 9, or 10 ; it refers to core memory if $C < 3$ or $C = 3$ and $B = 8$; it refers to fast memory if $C = 3$ and $B < 8$; and Advanced Control constructs an operand from the N-address if $C = 3$ and $B = 9$ or 10 or supplies the (zero) operand if $C = 3$, $B = 15$. If $C = 1$ counting is performed on the modifier register specified. Since there are 16 active modifier registers, and not 15, the case $C = 3$, $B = 8$ is necessary to specify a fixed memory location. The computer may be expected to run somewhat faster if short orders are used instead of long ones, and if registers in fast memory are used in preference to locations in the core memory.

For full-word store orders the core memory address or fast memory address specifies a destination rather than a source and the cases $C = 3$, $B = 1$, or $B \geq 9$ are illegal. (In the description of the fast memory it was stated that it was illegal to store into F1, so $C = 3$, $B = 1$ is illegal here. For $C = 3$, $B \geq 9$ an operand destination is meaningless.)

For exponent arithmetic, the "core address" defined above is not used to go to core memory but rather is reduced to 8 bits and combined with the exponent. More exactly, a word consisting of 4 copies of the address is placed in the ID register and Delayed Control combines arithmetically the right hand eight bits of this word with the exponent. Shift orders are also included in this class; however, only the rightmost 7 bits are used to define the number of shifts. The cases C = 3, B $\neq$ 8 are illegal for exponent arithmetic orders and shift orders.

For the quarter-word store orders, the B digits define the modifier register destination. These orders cannot refer to core memory and C is irrelevant. The instruction SIA should have B = one of 0, 4, 8, 12 because the integer will appear in the first 13 bits of the OUT register. The instruction SEX should have B = one of 3, 7, 11, 15 because the exponent will appear in the last 13 bits of the OUT register. If other B combinations occur they are not called illegal by the computer and might just be useful. For example, SIA, M1 would cause the 13 bits immediately to the right of the radix point to be stored in modifier M1.

The Floating Point Accumulator

There are a number of registers in the arithmetic unit whose action is required in the execution of instructions, which need not be described in the order code because results do not end up there. For one order, SRM, we shall have to refer to some of these extra registers, but otherwise the description will center around the basic registers which hold the results of each instruction.

Accordingly, the accumulator consists of 3 registers A, Q, E. A holds 45 bits called $a_0, \ldots, a_{44}$ in 2s complement notation, with value

$$a = -a_0 + \sum_{i \neq 0} 2^{-i} a_i \text{ , so } -1 \leq a \leq 1-2^{-44}$$

Q holds 44 bits with significance to the right of A. These bits are called $q_{-1}, q_0, q_1, \ldots, q_{42}$ and have value

$$q = \sum 2^{-i-2} q_i \text{ , so } 0 \leq q \leq 1-2^{-44}$$

E holds 8 bits called $e_7, e_6, \ldots, e_0$ with integer value

$$e = -128\, e_7 + \sum_{i \neq 7} 2^i e_i \quad \text{so} \quad -128 \leq e \leq 127.$$

If a calculated e falls outside this range it is held modulo 256. A, Q are connected together as an 89-bit shifting register AQ which holds the fractional part

$$f = a + 4^{-22} q \text{ of the accumulator.}$$

Note that this means that the digit q_{-1} follows immediately after a_{44} and that Q has no sign digit. Shifts are base 4 only (2 binary places at a time), and the exponent e signifies a power of 4. The accumulator holds the number $n = f \cdot 4^e$.

A word W in memory (core or fast) consists of a 45-bit fraction x followed by a 7-bit exponent y at the right-hand end of the word. Its value is $w = x \cdot 4^y$, $-64 \leq y \leq 63$, and fields x, y are represented in 2s complement notation. Note that the range of exponents permitted in the accumulator is about twice that in memory, and the accumulator holds a double precision number.

Zero and Overflow

The representation in memory of a floating point zero is 0.4^{-64}, i.e., zero fractional part and the most negative exponent possible, and it is the only floating point number with this exponent. When -64 is detected as the exponent of an operand, some orders such as ADD (see later) are by-passed. In the accumulator, a zero indicator Z is turned on whenever $f = 0$ or when a calculated exponent is less than -128. The contents of the floating point accumulator is not otherwise altered (it is not cleared to a fixed value) so the numerical value of the accumulator contents depends on Z as well as the contents of A, Q, E. Whenever f is changed, Z is cleared. Store orders, logical shift orders, and orders which are by-passed do not clear Z. When the operand of certain arithmetic orders have exponent equal to -64, the arithmetic is not done and the order is by-passed.

An overflow indicator OV is turned on whenever any result is too large to be correctly represented, and remains on until cleared by a special jump-on-overflow order (JDC with B = 10 or 11). If Z is on, the setting of OV is inhibited except for the inverse divide order (VID), in which case the memory

operand divided by the zero accumulator contents is judged to be an overflowed number.

In floating point arithmetic, overflow of the fractional part is corrected by a right shift of AQ (division by 4) and the addition of 1 to the exponent. In logical shifts and double A (DBA) the loss of digits at the left end of A is considered normal. Therefore, for non-store orders, OV is set only if e exceeds 127 or if we are asked to divide by zero.

For store orders one may be required to supply a particular representation of the number, and in this case it turns out that either the fraction or the exponent may overflow the more restricted range of numbers permitted in the memory. In this case OV is also set.

Note that Z gives a continuous indication of whether the accumulator now holds zero, whereas OV is a cumulative indicator telling whether any result has exceeded range since OV was last reset.

The floating point store orders (including STF: store fixed point) conform to the convention on zero numbers in memory, in that if Z is on, or $e \leq -64$, or the 45-bit fraction to be stored consists of all zeros, then the number $0 \cdot 4^{-64}$ (absolute zero) is transferred to memory.

The conditions Z on or $y = -64$ or $x = 0$ affect the following orders:

Order	Condition	Action
ADD/SUB	$y = -64$	by-pass the order
ADD/SUB	z on and $y \neq -64$	obey "clear add"/"clear subtract"
MPY, Z on		by-pass the order
MPY, Z off, $y = -64$		partial normalize (see later), then by-pass the order
DIV, $x = 0$		set OV and by-pass the order
DIV, z on, and $x \neq 0$		set remainder = 0 and by-pass the order

When OV has been set, the results in the accumulator are judged wrong, and no attempt is made to maintain a consistent representation of wrong numbers. The orders STL, STQ, SEX are logical in nature. (They allow the programmer to store [illegible] digits in the accumulator without having any floating point conventions imposed on him). If he later uses such a number as a floating point operand it may have exponent -64 and non-zero fractional part.

Normalization

A number $p \cdot 4^q$ is called normalized if one of

(a) Z on

(b) $p = 0$

(c) $-1 \leq p < -\frac{1}{4}$

(d) $\frac{1}{4} \leq p < 1$

holds. Since $p \cdot 4^q = (4p) \cdot 4^{q-1}$ a small fraction p may be normalized by repeated left shifts provided one subtracts 1 from the exponent q for every left shift required. Note that the Z indicator can come on during normalization due to exponent underflow. Except for divide, the results of arithmetic operations are left un-normalized; however, the accumulator may be normalized at the <u>start</u> of multiply, divide, difference absolute value (DAV) and certain of the store orders.

Addition and Subtraction

The sum $x \cdot 4^y$ and $f \cdot 4^e$ is obtained with an error of at most 2^{-88} in its fractional part as follows.

(a) If $|e-y| > 44$, the sum is taken to be the number with the larger exponent.

(b) If $|e-y| \leq 44$, the fractional part of the number with smaller exponent is right-shifted $|e-y|$ base 4 positions and its first 89 bits (including sign digit) are added to the other fraction. The error is a truncation error to the right of the 89th bit. The larger exponent is assigned to the result.

Note: Some cases of floating point addition can take a large number of steps by the computer, and a correspondingly long time to execute the instruction. Sometimes these long add or subtract orders can be avoided by careful programming. Relative times for addition can be estimated from the number of steps as follows:

Case 1	$e-y \leq -45$	obey Clear Add	6 steps.
Case 2	$-44 \leq e-y \leq 0$,	about $5 + \lvert e-y \rvert$ steps	
Case 3	$1 \leq e-y \leq 22$	about $5 + 2\lvert e-y \rvert$ steps	
Case 4	$23 \leq e-y \leq 44$	about $-11 + \lvert e-y \rvert$ steps	
Case 5	$45 \leq e-y$	by-pass	3 steps.

[illegible]ions Z true always means Case 1, and Z false but y = [illegible] always means [illegible]

Multiplication

The accumulator is normalized, if necessary, and its first 45 bits are rounded to form a fraction a_r. The product $x \cdot a_r$ is formed in AQ and the sum of the two exponents is placed in E.

If q = 0 normalization is not necessary (and is not done), since the product $(a \cdot x)4^{e+y}$ is exact. Likewise if q becomes zero after some even number of base 4 shifts, multiplication begins at that point. Partial normalization may be described by these rules:

1. If Z is true or q = 0 or a is normalized we are done. Otherwise go to 2.
2. If one left shift (base 4) of AQ will normalize a, left shift one place and subtract 1 from the exponent. If this results in an exponent less than -128 set Z.
3. Otherwise left shift 2 places and subtract 2 from the exponent. If this results in an exponent less than -128 set Z. Now return to 1 above.

The same type of partial normalization is done at the beginning of the DAV instruction.

The rules for ordinary normalization follow:

1. If Z is true or a is normalized we are done. Otherwise go to 2.
2. If one left shift (base 4) of AQ will normalize a, left shift one place and subtract 1 from the exponent. If this results in an exponent less than -128 set Z.
3. Otherwise left shift 2 places and subtract 2 from the exponent. If this results in an exponent less than -128 set Z. Now return to 1.

Division

First, the accumulator is normalized. Then the number from memory is normalized. If the latter has a zero fractional part, OV is set and the order is by-passed at this point. Then if Z is true or the difference of exponents (i.e., the exponent to be assigned to the quotient) is less than -128, the remainder is cleared to floating point zero, Z is set and the order is by-passed. Otherwise the order is obeyed, and a quotient is formed which is either normalized or has fractional part $-\frac{1}{4}$ and is correctly rounded to 45 bits. After divide A holds the fractional part of the quotient, Q holds zero, and E holds the exponent.

If the Delayed Control order immediately following divide is SR (store remainder), the remainder from division which was held in other registers in the arithmetic unit called R, ES is transferred to memory. The remainder obeys the floating point zero convention for numbers to be stored.

Note that divide can produce exponent overflow.

We might call an <u>improper division</u> one in which the normalized divisor has a fractional part smaller in magnitude than the fractional part of the original dividend (before normalization). In this case 47 bits would be required to express the fractional part of the remainder. The first 45 of these are retained, and the two others agree with the 88th and 89th bits of the dividend.

<u>Round-Off</u>

The first 46 bits of the fractional part of the normalized infinite quotient are rounded to 45 bits by adding a 1 to the 46-bit position, letting carries propagate, and then truncating the result after 45 bits. If the resulting fraction is +1 it is replaced by $+\frac{1}{4}$, and 1 is added to the calculated exponent. If this addition of 1 causes the exponent to become equal to +128, then OV is set.

For orders other than divide, a different procedure is used to obtain the rounded value of f, namely a_r as follows:

$$a_r = a \qquad \text{if} \quad q < \frac{1}{2}$$

$$a_r = a + 2^{-44}a_{44} \quad \text{if} \quad q = \frac{1}{2} \; *$$

$$a_r = a + 2^{-44} \qquad \text{if} \quad q > \frac{1}{2}.$$

*This corresponds to the rule in decimal arithmetic that to round off a 5 choose the nearest even digit, for example, (.325) rounded = .32 whereas (.335) rounded = .34.

Values +1, $-\frac{1}{4}$, of a_r can occur even if f has been normalized. In multiply, inverse divide, and difference absolute value, these values of a_r are used in the arithmetic unit without additional normalization, since it has a somewhat wider range of numbers it may compute with during the execution of an instruction. In the case of store orders, the accumulator is not changed after round-off, but the rounded result may be renormalized on the way to the FO register.

Correct Overflow and Detect Zero

As has been described already, during the execution of an instruction, the exponent is monitored, and Z or OV is set if the exponent of the accumulator falls outside of the range $-128 \leq e \leq 127$. At the end of each instruction which affects the contents of the accumulator, the operation "correct overflow and detect zero" or (K) for short is performed, whose rules follow:

(1) If $a = q = 0$ set Z. If Z is set disregard (2) and (3)

(2) If $-1 > a$ or $a \geq 1$ right shift AQ by 1 place and add 1 to the exponent.

(3) If (2) results in exponent overflow set OV.

We now define zero to be a non-negative number even though the sign digit of the accumulator may be 1 when Z is set. The normal sign test of the accumulator will be "Jump if $a_o = 0$ or if Z is true" and its inverse is "Jump if neither is true."

Orders will be listed as mnemonic, followed by the sexadecimal + octal code, followed by name and description.

CAD (82) Clear Add. Replace A, Q, E by x, o, y. Z is cleared but would be set by (K) if $x = 0$.

CSB (80) Clear Subtract. Replace A, Q, E by -x, 0, y. Z is cleared but would be set by (K) if $x = 0$. If $x = -1$ then (K) replaces f, y by $\frac{1}{4}$, y + 1. This could not cause OV to be set since $y + 1 \leq 64$.

CAT (83) Clear Add Twice. Replace A, Q, E by 2x, 0, y. Z is cleared. If $-\frac{1}{2} > x$ or $x \geq \frac{1}{2}$ then (K) replaces f, y by $\frac{1}{4}$ f, y + 1. If $x = 0$, Z is set, and (K) cannot set OV.

CST (81) Clear Subtract Twice. Replace A, Q, E by -2x, 0, y. Z is cleared. If $-\frac{1}{2} \geq x$ or $x > \frac{1}{2}$ then (K) replaces f, y by $\frac{1}{4}$ f, y + 1. If $x = 0$, Z is set and (K) cannot set OV.

AND (85) Digitwise Logical Multiply. Clear Z. Replace the digits of A with digits consisting of the product $a_i \cdot x_i$ for each i. Do not change q or e. (K) may set Z if $a = q = 0$ at the end of this instruction. OV cannot be set nor can corrective right shifts be done by (K) .

LOR ([illegible]) <u>Digitwise Logical OR</u>. Clear Z. Replace the digits of A with digits consisting of 1's wherever a_i and x_i are not simultaneously zeros. Do not change q or e. (K) may set Z if a = q = 0 at the end of this instruction. OV cannot be set nor can corrective right shifts be done by (K).

NOT (84) <u>Clear Add Digitwise Complement</u>. Clear Z and Q. Replace a_i by digits $1-x_i$ in every digital position. Replace E by y. (K) will set Z if x is composed entirely of 1's, OV cannot be set nor can corrective right shifts be done by (K).

DBA (87) <u>Single Binary Logical Left Shift of A</u>. This is <u>not</u> an arithmetic order unless the result is in range. Replace a by 2a mod 2. Do not change Q or E. An operand is required for this instruction but it is not used, so C = 3, B = 15 is recommended because it is then a short instruction which does not hold up Advanced Control. Z is cleared and would be set by (K) if q = 0 and a was 0 or -1. This order can change the sign of the accumulator.

AND, LOR, and DBA are logical and since they reset Z without replacing the entire contents of the accumulator should not be used in floating point arithmetic.

ADD (92) Form the sum of $x \cdot 4^y + f \cdot 4^e$ as described on page 15. Apart from the cases Z true or y = -64, before (K) the accumulator will contain the double precision sum with exponent equal to the larger of the exponents of the two operands. (K) may set Z or may right shift one place, adding 1 to the exponent. This cannot cause OV to be set, since the resulting exponent will not exceed +127. Note that no automatic normalization is done during addition, so it can serve as both floating point addition and fixed point addition. The decision on whether to normalize is made at the time of a store order, and depends on the type of store order given.

SUB (90) Form $(-x) \cdot 4^y + f \cdot 4^e$ in a manner precisely analogous to ADD just above.

MPY (+0) <u>Multiply</u>. Partially normalise $f \cdot 4^e$ and call the result $f \cdot 4^e$. Then, if either Z is true or $y = -64$, set Z and bypass the order. Otherwise replace f by $x \cdot f$ in AQ and e by $e + y$ in E. If $e + y < -128$ set Z, and if $e + y \geq 128$ set OV. Then obey (K) which will set Z if x was zero and will right shift AQ one place and add 1 to the exponent if and only if $f = -1$ and $x = -1$. In this case (K) would set OV only if $e + y = 127$ before (K).

DIV (+1) <u>Divide</u>. Normalise $f \cdot 4^e$ and call result $f \cdot 4^e$. Normalise $x \cdot 4^y$ and call result $x \cdot 4^y$. If $x = 0$, set OV and bypass the order. If $x \neq 0$ and Z is true set remainder equal to zero and bypass the order. Form $(\frac{f}{x})$ rounded or $(\frac{f}{4x})$ rounded in A, set $Q = 0$ and set $E = e - y$ or $e - y + 1$ respectively. The remainder will have an exponent approximately 22 less than e unless it is precisely zero. OV or Z may be set if $e - y$ or $e - y - 1$ go outside the range -128 to +127. (K) will have effect only if $a = +1$. In this case the fractional part of the quotient is right shifted one place and 1 is added to the exponent. If exponent overflow results, OV is set.

NDV (+2) <u>Negative Divide</u>. Identical to DIV except that the divisor is $(-x) \cdot 4^y$.

VID (+3) <u>Inverse Divide</u>. The accumulator is normalised and rounded and a_r and x are interchanged and Q is cleared; e and y are also interchanged. If Z is true OV is set and the order bypassed at this point. Otherwise $x \cdot 4^y$ is normalised. If then $x = 0$, Z is set at this point and the order is bypassed. Otherwise division proceeds from this point, forming $\left(\frac{x}{a_r}\right) \cdot 4^{y-e}$ or $\left(\frac{x}{4a_r}\right) \cdot 4^{y-e+1}$ in a manner analogous to that described under DIV above.

LDQ (a1) <u>Load Q</u>. Z is cleared and the digits $q_{-1}\, q_0 \cdots q_{42}$ are set equal to the non-sign digits of x namely $x_1 x_2 \cdots x_{44}$. (K) might set Z if $a = 0$ and the non-sign digits of x were all zeros, but (K) could have no other effect. This is a <u>logical</u> order and should not be used in floating point programs.

DAV (a2) Difference Absolute Value. The accumulator is partially normalized and $a_r \cdot 4^e$ is formed. It is noted whether or not Z is true at this time. Then Z is cleared and $-|x \cdot 4^y|$ is placed in the accumulator as if by a CAD or CSB order. If now Z was true enter Ⓚ with action like that in CAD or CSB. Otherwise enter ADD or SUB to form $|a_r \cdot 4^e| - |x \cdot 4^y|$. Action from this point on is identical to the ADD or SUB order.

STR (+4) Normalize Round and Store. If Z is on, store 0.4^{-64} in FO and, subsequently in the memory location specified. Otherwise normalize. If Z is now true, store 0.4^{-64}. From this point on the accumulator is not changed, but the number stored may be changed. Form a_r. This may still be normalized or it may take on the undesired values $+1, -\frac{1}{4}$. In the latter two cases change this to $+\frac{1}{4}$, -1 and respectively add 1 to the exponent or subtract 1 from the exponent. If now the exponent is ≤ -64 store floating point zero. If the exponent is $\geq +64$ set OV. If floating point zero is not stored, store the number obtained by the above operations. Since only a 7-bit exponent is stored, if OV is set the exponent is stored modulo 128. Ⓚ will have no effect.

XCH (+5) The new value of the memory register is the same as if STR were executed. The new value of the accumulator is the same as if CAD were executed. Note that for C = 3 the case B = 1 is illegal for this order.

STN (+7) The accumulator is normalized and $-a_r$ is transferred to A and Q is cleared. Then STR is executed and it works OK even if $-a_r = +1$. Following STR, in this case Ⓚ would right-shift and add 1 to e.

STF (-0) Store Fixed Point Rounded. A fixed point number is either 0.4^{-64} or has exponent zero. If Z is not on, $f \cdot 4^e$ is converted to fixed point by shifting right or left and counting up or down respectively on e until e becomes zero. If overflow in the fractional part occurs, OV is set and the process continues. After the shift the accumulator is unchanged. a_r is formed. If it is equal to +1, OV is set. If Z is true or $a_r = 0$, 0.4^{-64} is stored. Otherwise $a_r \cdot 4^0$ is stored with the exception that +1 is stored as -1. The net result that numbers are stored modulo 2 except for zero is fixed point representation. Ⓚ can set Z if the result is a = q = 0.

STU (-4) Store Unnormalized but Rounded. Identical to STR except that there is no preliminary normalization. The use of this order at key places in a program may considerably reduce the number of shifts prior to store orders and prior to succeeding add or subtract orders.

SRM (a3) Store Remainder. If this order follows a DIV, NDV or VID order with no intervening Delayed Control orders, the remainder (unnormalized, unrounded, but correctly represented even if zero) is stored from the R, ES registers. Since R, ES are used by most instructions, the use of this order following a non-divide order might be useful for engineering routines, but a catalogue of results expected would be voluminous. SRM sets overflow if the remainder has an exponent ≥ 64, which can happen even when the quotient is in range. (K) has no effect.

STC (+6) Store Clear. The accumulator is not cleared. The first part of this instruction coincides with STR. Suppose $a_r = a + 2^{-44}\epsilon$ where $\epsilon = 0$ or 1. Then $f \cdot 4^e = (a + 4^{-22}q) \cdot 4^e = [a_r + 4^{-22}(q-\epsilon)] \cdot 4^e = a_r \cdot 4^e + (q-\epsilon)4^{e-22}$. Now STR stores a number numerically equal to $a_r \cdot 4^e$. STC after doing this, transfers $q-\epsilon$ to A, clears Q to zero and subtracts 22 from e so the accumulator holds the remainder from the store operation. (K) may set Z if $q-\epsilon = 0$ or if $e - 22 < -128$. This order allows a double precision representation in memory in which the most significant half is correctly rounded, and, if STU follows STC, the least significant half has an exponent nearly always 22 less than the most significant half. The exceptions to this rule are when the most significant half rounds to +1 or $-\frac{1}{4}$, or when the least significant half is judged zero.

ASC (96) Add and Store Clear. Identical in effect to ADD followed by STC. Note that C = 3 and B = 1 is illegal.

SSC (94) Subtract and Store Clear. Identical in effect to SUB followed by STC. Note that C = 3 and B = 1 is illegal.

SIF (-1) Store Integer Part as a Floating Point Number. If Z is not true, the accumulator is shifted (see STF) until its exponent becomes equal to +22. This means that the radix point lies 44 bits to the right of a_0, that is, it lies between A and Q. If, during this process, overflow of the fractional part occurs, OV is set. Then if

a = 0 or Z is true, $0 \cdot 4^{-64}$ is sent to memory. Otherwise $a \cdot 4^{22}$ which is the integer part of the number, modulo 2^{45}, is sent to memory. Then (K) is obeyed, and it might set Z. Note that if $+2^{45}$ is the correct answer, the accumulator will have -2^{45}, OV will have been set, and (K) will not do a corrective right shift.

STL (-5) Store Logical. $a \cdot 4^{e}$ is transferred to memory, regardless of Z. OV is not set, a is not changed, and e is stored modulo 128. This is not a floating point order.

STQ (-6) Store Q. $q \cdot 4^{e}$ is transferred to memory, regardless of Z. OV is not set, a, q are not changed, and e is stored modulo 128. This is not a floating point order.

SEQ (-2) Store Rounded with Exponent Equal. This order has no operand, but an operand is implied in Fl. Normally, the previous order would have been "Load IN" (LIN) to be described later, but this is not necessary. Whatever number is in Fl at the time SEQ is obeyed furnishes a 7-bit exponent, and it is required to shift the accumulator in a manner exactly analogous to STF until its exponent becomes equal to this number and then round-off and store with qualifications identical to STF.

For the next group of orders, Advanced Control places the same address into all four quarters of Fl. This address is interpreted modulo 256 for the first four and modulo 128 for ARS, LRS, and is called y^* and y respectively.

CAE (97) Clear Add Exponent. Place y^* in E. (K) has no effect.

CSE (95) Clear Subtract Exponent. Place $-y^*$ in E. If $y^* = -128$ set OV.

ADE (93) Add to Exponent. Place $e + y^*$ in E. Set OV or Z if the range $-128 \leq \text{exponent} \leq 127$ is exceeded, but do not set OV if Z is true.

SBE (91) Subtract from Exponent. Place $e - y^*$ in E. Set OV or Z if the range $-128 \leq \text{exponent} \leq 127$ is exceeded, but do not set OV if Z is true.

[illegible] (87) Logical Right Shift A. If y is positive, translate the digits of the A register right 2y bits without sign digit duplication, and throw away those that pass the right hand end of the A register. If y is negative, translate the digits of the A register left, throwing away those that pass the left end of the register. Do not set OV. (K) could set Z. This is a logical order which would not be useful in floating point programs. Note: 2y bits are y base 4 shifts

LRS (a5) Long Logical Right Shift. The double length equivalent of ARS above. This is also a base 4 shift.

Of the Delayed Control orders, there remain only two store orders which produce 13-bit results to be transferred to modifier registers in the fast memory.

SEX (-7) Store Exponent. A word whose first 39 digits agree with the first 39 digits of a and whose last 13 digits agree with the 8-bit exponent e extended 5 digits left by duplication of the sign digit, is placed in F0, then the $\frac{1}{4}$ word of F0 aligned with M_B is copied into M_B. If the modifier register specified is M3, M7, M11 or M15, the last $\frac{1}{4}$, i.e., the exponent, is stored in it. This is a logical order and Z is disregarded.

SIA (-3) Store Integer Address. The accumulator is shifted until its exponent is equal to +6 in a manner similar to SIF. If th base 4 exponent is +6 the radix point lies between the 13th and 14th digits of the A register. OV or Z may be set during the shift but if Z was true beforehand, no shift is made. Now if a = 0 or Z is true, 0.4^{-64} is placed in F0. Otherwise $a \cdot 4^e$ is placed in F0. If the modifier register specified is one of M0, M4, M8 or M12 the first quarter word, i.e., the integer part of the accumulator, modulo 2^{13} is copied into the M_B. Otherwise a different quarter word is copied (see SEX).

4. Orders Which Do Not Involve Floating Point

The orders described in the last section are obeyed first by Advanced Control, which obtains any needed operand and places it in F1, then by Delayed Control which performs any necessary floating point arithmetic, and then, in the case of store orders, by Advanced Control again. SIA and SEX were a special type: always short, B represents the modifier, and C is irrelevant. Otherwise address construction was fairly uniform: if C < 3 or C = 3, B = 8 an address is constructed and depending on the order type this either defines a core memory location, or the address is quadruplicated and used. Let us refer to this process as normal address construction. For floating point orders additional options were provided: C = 3 and B <8 means fast register F_B with the proviso that F1 isn't a destination,

C = 3, B = 1 is illegal for certain orders. Likewise floating point operands were generated for the cases C = 3, B $\geq$ 9. These additional options do not apply to the next class of orders: Advanced Control and Interplay orders with normal address construction. If C = 3 and B $\neq$ 8 these orders are illegal.

PID (23) Prepare Input Device. After the address is constructed it is sent to Interplay and is interpreted as a 5-bit field at the left-hand end signifying the channel number and an 8-bit field specifying details of a block transfer into the core memory. At the time of this writing, channel 0 has been assigned to the drum and the 8-bit field represents the drum block. No other assignments have been made yet.

POD (62) Prepare Output Device. Similar to PID above except that a transfer from the core memory is intended.

IBT (22) Initiate Block Transfer. The address constructed specifies the core address at which the transfer will start. Depending on the device in question, the transfer will cease when a stop character is reached or when the core address reaches an address 1 less than the next multiple of 256, whichever happens sooner. In the case of the drum, and probably all devices which operate on a fixed 256-word block, only the first 5 bits of this address matter--the others are replaced by zeros so a drum transfer always begins at a core address which is a multiple of 256 and ends at the address 1 less than the next multiple of 256.

PID, POD, IBT are the only orders obeyed by Interplay.

BBF (43) Busy Block Flipflop. The first five bits of the address constructed define a block in core memory whose indicator is to be set to the "busy" state.

FBF (42) Free Block Flipflop. The first five bits of the address constructed define a block in core memory whose indicator is to be set to the "free" state.

Note: For the summer of 1962 the 5 orders PID, POD, IBT, BBF, and FBF cannot be obeyed since Interplay and the Block Checker have not been built yet. A console switch can be set so either these orders are always legal, or they are legal only in the interrupt mode.

LIN (60) Load IN Register. Four copies of the constructed address are placed in F1. This order usually precedes SEQ.

JLH (54) Jump to Left-Hand Control Group. A "jump" instruction is a control transfer or branch operation. JLH has the effect that the next order obeyed begins at the first control group of the core word defined by the address. This is an unconditional jump which lacks generality and is mainly useful in returning from a subroutine.

ATN (21) Add to Next Address. Certain orders, of which this is the first example, influence the address construction of order following. The address formed by this instruction is added to the address of the next instruction. ATN may be repeated. Example. Suppose one wished to copy into the accumulator the number from the memory location defined by the sum of modifiers M2, M5, M7. The program is

```
ATN 2,0
ATN 5,0
CAD 7,0   or 3 short orders.
```

The first instruction adds (M2) to the second. The second would normally add (M5) to the 3rd, but since it has (M2) added to its address it therefore adds (M2) + (M5) to the 3rd instruction. The 3rd instruction would normally have core address (M7) but this is increased by (M2) + (M5) making a total of (M2) + (M5) + (M7) as required.

SFN (41) Subtract from Next Address. The address formed is subtracted from the address of the next instruction.

ORB (61) Logical OR with B Digits of the Next Instruction. The rightmost 4 bits of the address constructed are combined with the 4 B-bits of the instruction following--the next instruction will have zeros only in those bit positions of the B field where both bits are zero. This is a useful instruction for breaking up words into quarter-words but is not very useful for combining quarter words into words.

This completes the list of instructions for which normal address construction applies. The next group of instructions are always short.

ASN (32) Add Special Register to Next Address. This is a short instruction. Provision is made in the computer for up to 64, 13-bit registers called special registers. For the summer of 1962 only one is available, and it represents the next paper tape character, that is, 6 zeros followed by a 7-bit character. For this instruction the B and C fields are combined into one field (numerically 4B + C) which specifies a special register numbered 0 to 63. Temporarily any B, C combination calls for paper tape, but later on only the 0,0 combination will specify paper tape. ASN 0,0 causes a character to be read from paper tape and added to the address of the next instruction.

SSN (72) Subtract Special Register from Next Address. This is a short instruction. Similar to ASN except that subtraction rather than addition is done.

SSR (73) Store in Special Register. This is a short instruction, and its address is zero unless SSR was preceded by an "add to next" type order. The B,C fields specify which special register the address is to be stored in. If an instruction such as ATN had preceded this instruction, then, in general, something other than zero would be stored in the special register. Initially SSR refers to the paper tape punch regardless of the values of B,C but later on only 0,0 will refer to the punch. Digits punched are the rightmost 7 bits of the address.

CJF (57) Count and Conditional Jump to First. This is a short instruction. One is added to the contents of modifier M_B and the result is returned to M_B. If the result is non-zero, jump to the instruction at F8 position C, (C = 0,1,2, or 3), otherwise obey the next instruction in sequence. The purpose of this instruction is to be able to obey simple loops of instructions inside F8 and F9 if the loop condition is just a count. In these cases the instruction words are read from memory just once and are held in F8 and F9 for repeated execution. F8 contains the contents of an even-numbered core memory location, and F9 holds the contents of the next higher numbered location, which is odd. Note that this implies that the normal method of counting is to place the negative of the count in a modifier register and count up to zero. For a long

program consisting of long and short instructions intermixed, the programmer would refer to instructions symbolically and would not, in general, know what word and position any instruction occupied. One of the operations which the assembly routine must be able to do is to insert a jump to the left-hand control group of the next even address so these short loops may be correctly positioned. During the period when there is only one core memory CJF will have the effect of conditionally jumping to F9, position C which means jump to position C of the word containing the present instruction.

CJS (55) Count and Conditional Jump to Second. This is a short instruction, whose action is similar to CJF above except the destination is F9 position C. During the period when there is only one core memory the action of CJS is identical to the action of CJF.

HLT (63) Black Switch Conditional Stop. This is a short instruction. A console switch, colored black has 2 stable positions--the center position and the down position. The up position is spring loaded. In the down position HLT orders are ignored. In the center position an HLT order causes the computer to stop prior to executing the next instruction following the HLT instruction. If the black switch is raised to the up position and then released, this HLT is by-passed and the computer will stop on the next HLT instruction. The B and C digits have no effect.

The next group of instructions are always long, and N, the second control group, specifies the address.

CJU (77) Count and Jump if the Result is Unequal to Zero. Long. Add 1 to (M_B) and return the result to modifier M_B. If it is non-zero jump to word N, position C, otherwise obey the next instruction in sequence.

CJZ (37) Count and Jump if the Result is Zero. Long. Add 1 to (M_B) and return the result to modifier M_B. If it is zero, jump to word N, position C, otherwise obey the next instruction in sequence. This is a very rarely used instruction--CJU would be much more common in programs.

JPM (74) Jump if Positive Modifier. Long. If the leftmost of the 13 bits of (M_B) is a 0, jump to N, position C, otherwise obey the next instruction in sequence. We may regard the integer held in M_B as either lying in the range $-4096 \leq (M_B) \leq 4095$ if the leftmost digit is regarded as having negative weight, or lying in the range 0 to 8191 for core addresses, or -8191 to 0 for orders like CJF, CJS, CJU, CJZ.

JNM (34) Jump if Negative Modifier. Long. If the leftmost of the 13 bits of (M_B) is a 1, jump to word N, position C. Otherwise obey the next instruction in sequence.

JZM (35) Jump if Zero Modifier. Long. If all 13 bits of (M_B) are zeros, jump to word N, position C. Otherwise obey the next instruction in sequence.

JUM (75) Jump if Modifier is Unequal to Zero. Long. If the 13 bits of (M_B) are not identically zero, jump to word N, position C. Otherwise obey the next instruction in sequence.

JSB (76) Jump to Subroutine. Long. Let H be the location of the N address of this JSB instruction. Place H + 1 into M_B and jump to word N, position C. Conventionally B = 15 and the subroutine returns control to the left-hand control group of the word following the JSB instruction. Thus entry to a subroutine at location I would be accomplished by JSB, 15, 0, I and return from subroutine would be accomplished by JLH, 15, 0.

JDC (56) Jump on One of a Diversity of Conditions. Long. The B field specifies one of 16 possible conditions to be tested. If the condition is true, the next instruction is obeyed from word N, position C. Otherwise obey the next instruction in sequence. The conditions are

B = 0 Unconditional. This also causes the computer to leave the interrupt mode if it happens to be in it.

B = 1 Don't jump. A long null order.

B = 2 Accumulator positive or zero (Z on or $a \geq 0$)

B = 3 Accumulator negative

B = 4 Accumulator unequal to zero (Z not on)

B = 5	Accumulator zero
B = 6	Accumulator positive and not zero
B = 7	Accumulator zero or negative
B = 8	OV on
B = 9	OV not on
B = 10	OV on, then clear OV in any case
B = 11	OV not on, then clear OV in any case
B = 12	Green switch center (another console switch)
B = 13	Green switch down (its normal position)
B = 14	Digit $a_0 = 0$ (Useful mainly in logical operations)
B = 15	Digit $a_0 = 1$ (Useful mainly in logical operations)

Note that if $B \neq 0, 1, 12,$ or 13 this jump is conditional on the arithmetic result after Delayed Control has finished any instruction in progress and quite possibly another instruction prepared by Advanced Control. Such JDC orders can greatly slow down the machine, and one of the objectives of good programming is to reduce the number of these, at least in critical parts of a program.

LDM (71) Load Modifier from Core Memory. Long. The quarter word aligned with M_B in word N in memory is copied into M_B. If B = 0, 4, 8 or 12 this would be the first quarter word, if B = 1, 5, 8, or 13 this would be the second quarter word, etc.

SQW (31) Store Quarter Word from Fast Memory to Core Memory. Long. Into core word N, position C, copy that quarter word of F_B which lines up with it. Note particularly that B is the name of a fast register, and not of a modifier. If B = 1 or $B \geq 8$ the instruction is illegal.

The remaining instructions are long or short depending on whether C = 2, 3, or 1,0. If we name the C bits C_1 C_2, then these instructions are long if and only if $C_1 = 1$. The address is 0 if the instruction is short, and N if the instruction is long.

LFR (70) Load Fast Register. Long if $C_1 = 1$. Copy the word from core location given by the address into F_B. If B = 0 or 1 or $B \geq 8$ the instruction is illegal. C_2 has no effect and should be conventionally zero.

SFR (30) Store Fast Register. Long if $C_1 = 1$. Copy the word from F_B into core location given by the address. If $B = 1$ or $B \geq 8$ the instruction is illegal. C_2 has no effect and should be conventionally zero.

Modifier Arithmetic

The remaining 12 instructions cause the contents of the modifier register M_B to be combined arithmetically with the address. If $C_1 = 0$ the instruction is short and the address is 0. If $C_1 = 1$ the instruction is long and the address is N. If $C_2 = 0$, the result of this arithmetic is returned to the register M_B. If $C_2 = 1$, the result is not returned to M_B, but is added to the address of the instruction following. The description of these instructions will assume that $C_2 = 0$ for simplicity in explanation.

CAM (27) Clear Add Modifier. Long if $C_1 = 1$. The address is copied into M_B.

CSM (25) Clear Subtract Modifier. Long if $C_1 = 1$. The negative of the address is formed and copied into M_B.

ADM (67) Add to Modifier. Long if $C_1 = 1$. The address is added to (M_B) and the result is returned to M_B.

SBM (65) Subtract from Modifier. Long if $C_1 = 1$. The address is subtracted from (M_B) and the result is returned to M_B.

CNM (24) Clear Negate Modifier. Long if $C_1 = 1$. The digitwise complement of the address is formed, and is copied into M_B. The digitwise complement of a binary number is the number consisting of zeros when the original number had ones, and vice versa. In this case, numerically, the digitwise complement is 8191-address.

CRM (26) Circular Right Shift Modifier. Long if $C_1 = 1$. The 4 rightmost bits of the address define a number of shifts p where $0 \leq p \leq 15$. The modifier contents (M_B) is rotated right circularly p places and then returned to M_B. Note that a shift of 13 places brings us back to where we started from so $p = 13$ has the same effect as $p = 0$

$p = 14$ has the same effect as $p = 1$

$p = 15$ has the same effect as $p = 2$.

ANM (47) AND with Modifier. Long if $C_1 = 1$. The 13 bits of the address are ANDed with the corresponding bits of (M_B) and the result is returned to M_B. A bit position of the result has a 1 if and only if both operands had ones in that digital position.

ORM (46) OR with Modifier. Long if $C_1 = 1$. The 13 bits of the address are ORed with the corresponding bits of (M_B) and the result is returned to M_B. A bit position of the result has a 1 if and only if at least one of the operands had a 1 in that digital position.

EOM (66) Exclusive OR with Modifier. Long if $C_1 = 1$. The exclusive OR (or addition without carries) of the 13 address bits and the 13 (M_B) bits is formed and returned to M_B. The result has ones in those bit positions in which the two operands differed.

EQM (64) Equivalence with Modifier. Long if $C_1 = 1$. The equivalence function of the address and (M_B) is formed in every digital position, and the result is returned to M_B. The result has ones in those bit positions in which the two operands agreed. equiv = $\sim\oplus$

NAM (45) Negate then AND with Modifier. Long if $C_1 = 1$. The digitwise complement of the address is formed, and ANDed digit by digit with (M_B). The result is returned to M_B. The result has ones only in those bit positions where the address had zeros and the modifier had ones.

NOM (44) Negate, then OR with Modifier. Long if $C_1 = 1$. The digitwise complement of the address is formed and ORed digit by digit with (M_B). The result is returned to M_B. The result has ones in those bit positions where the address had zeros or the modifier had ones, or both.

APPENDIX

TABLE I Address Construction

Normal: LDQ, CAD, CSB, CAT, CST, NOT, AND, LOR, DBA, ADD, SUB, MPY, DIV, NDV, VID, DAV

Normal, $C = 3$, $B = 1$ or $B \geq 9$ illegal: STR, STU, STN, STC, STF, SIF, SEQ, STL, STQ, SRM ASC, SSC, XCH

Normal, $C = 3$, $B \neq 8$ illegal, address used as operand: CAE, CSE, ADE, SBE, ARS, LRS, LIN, PID, POD, IBT, BBF, FBF, JLE, ATN, SFN, ORB

Short B means M_B: SIA, SEX, CJF, CJS

Short, $4B + C$ is name of special register: ASN, SSN, SSR

Long, C represents 1/4 W except for LDM: CJU, CJZ, JPM, JNM, JZM, JUM, JSB, LDM, SQM, JDC

$C_1 = 1$ Means Long, C_2 irrelevant: LFR, SFR

$C_1 = 1$ Means Long, $C_2 = 1$ Means add to next: CAM, CSM, ADM, SBM, CNM, CRM, ANM, ORM, EOM, NAM, NOM, EQM

Short, B and C don't mean anything: HLT

TABLE 2 Additional Information on Instructions

Instructions	Information
LDQ, CAD, CSB, CAT, CST, NOT, AND, LOR, DBA	Clear Z first
ADD, SUB, MPY, ASC, SSC	Special cases if Z is true or if $y = -64$
DIV, NDV, VID	Special cases if Z is true or if $x = 0$
STR, STU, STN, STC, STF, SIF, SEQ, SIA	Special case if Z is true $(0 \cdot 4^{-64})$
STL, STQ, SEX, SRM, ARS, LRS, LIN	Disregard Z
CAE, CSE, ADE, SBE	Z or OV may be set. If Z is true OV is not set
XCH	Special case if Z is true $(0 \cdot 4^{-64})$. Then clear Z
PID, POD, IBT, BBF, FBF	Don't work now. Console switch makes then legal or illegal as a group unless we are in the interrupt mode

TABLE 3. ORDER CODE INDEX

Order Codes		Page No.
ADD	92	13,14,19
ADE	93	10,23
ADM	67	31
AND	85	18,19
ANM	47	32
ARS	a7	23
ASC	96	22
ASN	32	27
ATN	21	26,27
BBF	43	25
CAD	82	7,14,18
CAE	97	23
CAM	27	7,31
CAT	83	18
CJF	57	6,27,29
CJS	55	6,28,29
CJU	77	28,29
CJZ	37	28,29
CNM	24	31
CRM	26	31
CSB	80	14,18
CSE	95	23
CSM	25	31
CST	81	18
DAV	a2	15,16,21
DBA	87	14,19
DIV	+1	14,20,22
EDM	66	32
EQM	64	32
FBF	42	25
HLT	63	28
IBT	22	25
JDC	56	9,10,13,29
JLH	54	26,29
JNM	34	29
JPM	74	29
JSB	76	29
JUM	75	29
JZM	35	29

Order Codes		Page Nos.
LDM	71	30
LDQ	a1	20
LFR	70	7,8,30
LIN	60	23,26
LOR	86	19
LRS	a5	23,24
MPY	+0	7,10,14,20
NAM	45	32
NDV	+2	20,22
NOM	44	32
NOT	84	19
ORB	61	26
ORM	46	32
PID	23	25
POD	62	25
SBE	91	23
SBM	65	31
SEQ	-2	23
SEX	-7	10,12,14,24
SFN	41	26
SFR	30	7,8,31
SIA	-3	10,12,24
SIF	-1	22
SQW	31	30
SRM	a3	22
SSC	94	22
SSN	72	27
SSR	73	27
STC	+6	22
STF	-0	14,21
STL	-5	14,23
STN	+7	21
STQ	-6	14,23
STR	+4	7,10,21
STU	-4	22
SUB	90	14,19
VID	+3	13,20,22
XCH	+5	21

NOTE: The page number where the order is defined in the test is underlined.

TABLE 4 Order Code Listed Numerically

First Sexadecimal Digit \ Second Octal Digit	0	1	2	3	4	5	6	7
2		ATN	IBT	PID	CNM	CSM	CRM	CAM
3	SFR	SQW	ASN		JNM	JZM		CJZ
4		SFN	FBF	BBF	NOM	NAM	ORM	ANM
5					JLH	CJS	JDC	CJF
6	LIN	ORB	POD	HLT	EQM	SBM	EOM	ADM
7	LFR	LDM	SSN	SSR	JPM	JUM	JSB	CJU
8	CSB	CST	CAD	CAT	NOT	AND	LOR	DBA
9	SUB	SBE	ADD	ADE	SSC	CSE	ASC	CAE
+	MPY	DIV	NDV	VID	STR	XCH	STC	STN
-	STF	SIF	SEQ	SIA	STU	STL	STQ	SEX
a		LDQ	DAV	SRM		LRS		ARS

NOTE: All unassigned orders are illegal, namely the blanks in this table and orders whose first sexadecimal digit is 0, 1, b, c, or d.