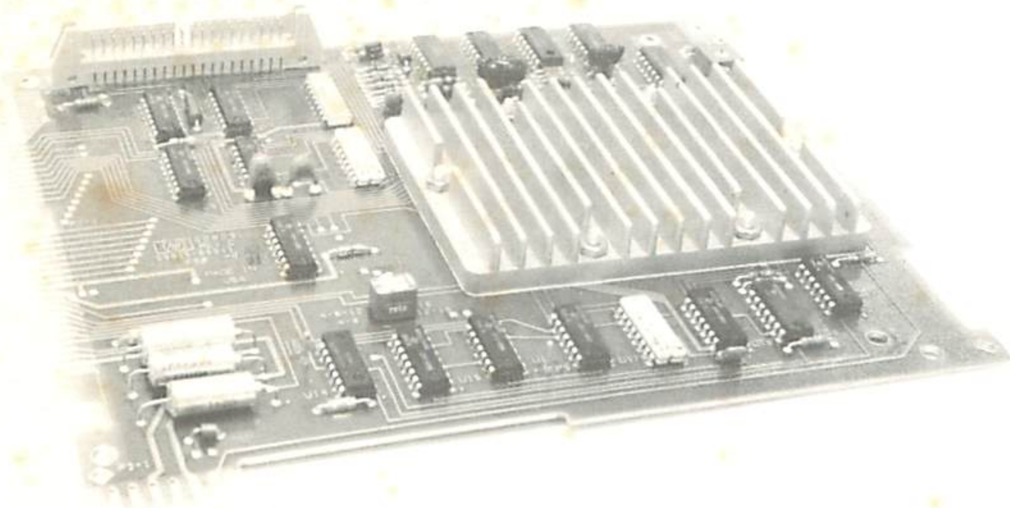
REFERENCE MANUAL
for the
CPD N-MOS II PROCESSOR



Calculator Products Division August 1976

TABLE OF CONTENTS

CHAPTERS

PREFACE . . . viii

PROCESSOR

DESCRIPTION OF THE PROCESSOR . . . 1
GENERAL INFORMATION . . . 2
MEMORY CONVENTIONS . . . 3
MEMORY CYCLES . . . 5
THE $\overline{\text{BYTE}}$ LINE . . . 6
RAL LINE . . . 7
FUNCTIONAL DESCRIPTION OF THE BPC . . . 8
INDIRECT ADDRESSING . . . 8
MULTI-LEVEL INDIRECT ADDRESSING . . . 8
SINGLE-LEVEL INDIRECT ADDRESSING . . . 9
MEMORY REFERENCE INSTRUCTIONS AND PAGE ADDRESSING . . . 9
ABSOLUTE ADDRESSING . . . 11
RELATIVE ADDRESSING . . . 11
BASE PAGE ADDRESSING . . . 11
CURRENT PAGE ADDRESSING . . . 12
SUBROUTINES . . . 15
FLAGS . . . 15
BUS REQUESTS AND INTERRUPTS . . . 15
FUNCTIONAL DESCRIPTION OF THE IOC . . . 18
GENERAL INFORMATION ABOUT I/O . . . 18
I/O BUS CYCLES . . . 18
STANDARD I/O . . . 21
ADDRESSING THE PERIPHERAL . . . 21
CHECKING STATUS . . . 21
INITIATING I/O BUS CYCLES . . . 21
THE ODDBALL POSSIBILITIES . . . 22
THE INTERRUPT SYSTEM . . . 23
PRIORITY . . . 23
INTERRUPT POLLS . . . 23
INTERRUPT TABLE . . . 24
INTERRUPT PROCESS SUMMARY . . . 26
INTERRUPT SERVICE ROUTINES . . . 26

TABLE OF CONTENTS

CHAPTERS

PROCESSOR

FUNCTIONAL DESCRIPTION OF THE IOC

THE INTERRUPT SYSTEM (CONT.)

HOW A PERIPHERAL KNOWS WHETHER TO USE INTERRUPT OR SIMPLE I/O . . . 28

BOMBPROOFING THE MAINLINE FIRMWARE . . . 28

"SIMULTANEOUS" ACTIVITIES . . . 28

WHEN TO CEASE INTERRUPT MODE OPERATION . . . 29

RETURNING FROM INTERRUPT SERVICE ROUTINES . . . 30

DISABLING THE INTERRUPT SYSTEM . . . 30

DIRECT MEMORY ACCESS . . . 31

ENABLING AND DISABLING THE DMA MODE . . . 31

REGISTER SET-UP . . . 32

DMA INITIATION . . . 32

DATA REQUEST AND TRANSFER . . . 33

DMA TERMINATION . . . 33

THE PULSE COUNT MODE . . . 33

PLACE AND WITHDRAW . . . 34

THE NOTATION OF A STACK . . . 34

STACK OPERATIONS . . . 34

PLACE AND WITHDRAW FOR BYTES . . . 35

INITIALIZATION OF TURN-ON . . . 38

GENERAL INFORMATION ABOUT THE EMC . . . 39

NOTATION . . . 39

DATA FORMAT . . . 40

A BEGINNER'S LOOK AT CALCULATOR ARITHMETIC . . . 41

NUMERICAL REPRESENTATIONS . . . 41

BINARY . . . 41

BINARY-CODED DECIMAL . . . 42

BINARY ARITHMETIC . . . 43

BINARY COMPLEMENTS . . . 43

TWO'S COMPLEMENT SUMMATION . . . 45

TWO'S COMPLEMENT SUBTRACTION . . . 45

TWO'S COMPLEMENT OVERFLOW . . . 50

TABLE OF CONTENTS

CHAPTERS

PROCESSOR

A BEGINNER'S LOOK AT CALCULATOR ARITHMETIC (CONT.)

- MULTI-PRECISION BINARY ARITHMETIC 52
- ARITHMETIC SHIFTS 53
- BINARY MULTIPLY 55
- BCD ARITHMETIC 55
  - DECIMAL CARRY 56
  - TEN'S COMPLEMENT FOR BCD 56
  - TEN'S COMPLEMENT ARITHMETIC DEMONSTRATION 59
- FLOATING-POINT SUMMATIONS 61
  - OFFSETS. 61
  - MANTISSA ADDITION. 62
- NORMALIZATION 63
- ROUNDING 63
- FLOATING-POINT MULTIPLICATION 64
- FLOATING-POINT BCD DIVISION 66
  - THE DIVISION ALGORITHM. 66
- THE FDV INSTRUCTION 68
  - SAMPLE DIVISION ROUTINE 71

INSTRUCTIONS

INTRODUCTION TO THE MACHINE INSTRUCTIONS. 1

- NOTATION 1

BPC MACHINE INSTRUCTIONS 2

- MEMORY REFERENCE GROUP 2
- SHIFT-ROTATE GROUP 4
- ALTER-SKIP GROUP. 5
- COMPLEMENT-EXECUTE GROUP 10

IOC MACHINE INSTRUCTIONS 12

- STACK GROUP. 12
- I/O GROUP 14
- INTERRUPT GROUP 14
- DMA GROUP 15

TABLE OF CONTENTS

CHAPTERS

INSTRUCTIONS

EMC MACHINE INSTRUCTIONS 16
THE FOUR WORD GROUP 16
THE MANTISSA SHIFT GROUP 16
THE ARITHMETIC GROUP 17

ASSEMBLER

INTRODUCTION TO THE ASSEMBLER 1
GENERAL INFORMATION 1
INSTRUCTION FORMAT 2
STATEMENT CHARACTERISTICS 2
LABEL FIELD 3
OPCODE FIELD. 4
OPERAND FIELD 5
SYMBOLIC TERMS 6
NUMERIC TERMS 8
THE ASTERISK. 8
EXPRESSIONS 8
INDIRECT ADDRESSING 9
BASE PAGE AND CURRENT PAGE ADDRESSING. 9
COMMENT FIELD 9
STATEMENT LENGTH 10
ASSEMBLER PSEUDO INSTRUCTIONS 11
ASSEMBLER CONTROL 11
ORG AND ORR 11
NEW INSTRUCTION DEFINITION 12
PARTITIONING A BINARY TAPE 14
CONDITIONAL ASSEMBLY 15
AUTOMATIC STATEMENT REPETITION 17
SOURCE TERMINATION 17
ADDRESS AND SYMBOL DEFINITION 18
CONSTANT DEFINITION 20
STORAGE ALLOCATION 23
ASSEMBLY LISTING CONTROL 23

TABLE OF CONTENTS

CHAPTERS

ASSEMBLER

ASSEMBLER INPUT AND OUTPUT 26
THE CONTROL STATEMENT 26
THE SOURCE PROGRAM 27
THE LISTING 27
BINARY OUTPUT 28

APPENDIX

APPENDIX 1
ASSEMBLER ERROR MESSAGES 1
BINARY LOADERS 3
OUTPUT PAPER TAPE FORMAT 5
ABSOLUTE BINARY OBJECT PROGRAM 5
ADDING PRE-DEFINED SYMBOLS TO ASMA 6
THE STRUCTURE OF THE ASSEMBLER 9
PSEUDO INSTRUCTIONS 11
MACHINE INSTRUCTIONS 12
INSTRUCTION BIT PATTERNS 18
MEMORY REFERENCE GROUP. 18
SHIFT-ROTATE GROUP 18
SKIP GROUP 19
RETURN GROUP. 19
COMPLEMENT GROUP 20
ALTER GROUP 20
EXECUTE GROUP 21
16-BIT IOC ONLY GROUP 21
STACK GROUP 22
INTERRUPT GROUP 22
DMA GROUP 22
FOUR WORD OPERATION GROUP 23
MANTISSA SHIFT GROUP 23
ARITHMETIC GROUP 23
15/16 BIT BPC CONSOLIDATED CODING SHEET 24
15/16 BIT IOC CONSOLIDATED CODING SHEET 24
15/16 BIT EMC CONSOLIDATED CODING SHEET 24

TABLE OF CONTENTS

CHAPTERS

APPENDIX

APPENDIX (CONT.)

HP CHARACTER SET 25
CHARACTER CODES 26
BPC INSTRUCTION EXECUTION TIMES 27
EMC INSTRUCTION EXECUTION TIMES 28
IOC INSTRUCTION EXECUTION TIMES 29
EXPLANATION OF BOOTH'S ALGORITHM 30

FIGURES

PROCESSOR

Figure P-1. Simplified Block Diagram of the Processor . . . 1
Figure P-2. Nature of the BIB's 3
Figure P-3. Simplified Read Memory Cycle 5
Figure P-4. Simplified Write Memory Cycle 6
Figure P-5. Base Page Description 10
Figure P-6. Relative Addressing 14
Figure P-7. Bus Request Protocol 16
Figure P-8. A Write I/O Bus Cycle 20
Figure P-9. A Read I/O Bus Cycle 20
Figure P-10. The Interrupt Table With 15-Bit or 16-Bit Addressing.25
Figure P-11. How Not To Use The Interrupt Table 25
Figure P-12. Sixteen-Bit Stack Pointer Addressing 36
Figure P-13. Floating-Point Data Format 40
Figure P-14. The Internal Floating-Point Representation of .003587219 (= 3.587219×10^{-3}) 43
Figure P-15. Multi-Word Binary Addition Using the Extend Register.52
Figure P-16. Two's Complements of Multi-Word Binary Numbers . .53
Figure P-17. Floating-Point Data Format 54

TABLE OF CONTENTS

TABLES

PROCESSOR

Table P-1. Addressable Registers 4

Table P-2. Current Page Absolute Addressing for Memory Reference Instructions . . .13

Table P-3. Comparison of Decimal, Binary, and Octal . . .41

ASSEMBLER

Table A-1. Symbols Pre-Defined by the Assembler 7

PREFACE

This book is the result of an extensive revision of the "CPD PROCESSOR" manual first issued in early 1975. Things have changed a bit since then, and the old manual was getting pretty shakey. The development of the 16-bit version of the processor provided the opportunity to revise the entire book.

First, this book covers both versions of the processor; one with 15-bit (32K) addressing, and the other with 16-bit (64K) addressing. The assembler (ASMA) described herein has also been updated to work with the 16-bit version.

Next, numerous mistaken and misleading explanations have been corrected. Also, the information relating to the general attributes and operation of the hardware has been collected together and organized as an introduction and overview of the entire processor. However, the book does not educate the reader in the general notion of what a processor is, or in the ins and outs of assembly language programming; it is still very helpful if one is familiar with the 2100-series computers.

As before, the book is aimed primarily at engineers and technicians within HP who will recognize the attributes of the processor and apply them to their own situation. Even so, there are still places where the explanation becomes detailed. The explanations of the interrupt process and of arithmetic are examples. There are other areas which the reader is simply expected to absorb on his own. The assembler is a good example; all the explanation in the world (and we give quite a bit) won't remove the need for a little bit of experience.

If you are a beginner, you probably shouldn't try to read the book from cover to cover, in the order given. It would be better if you mix your exposure to the system overview (at least skip the arithmetic), machine-instructions, and the assembler.

A comment on the section on arithmetic is in order. First, it would be impossible to understand the EMC arithmetic instructions without reference to some detailed examples. Second, its been my experience that typically there's one guy who sits in a corner, mutters out loud a lot, and who writes all the math routines. He's the only guy who knows how they work, and even he makes frequent references to the texts he used in school. And in general, if you ask three differenct people about some aspect of arithmetic, you'll get three different answers.

I don't suppose that too many people are really concerned about the nature of the EMC instruction set. But it needs explanation none the less. To do that, detailed examples are needed. To understand the examples, some familiarity with arithmetic techniques is needed. So I went the last mile and started at the beginning.

At present, there is exactly zero interfacing information that would allow a designer to create hardware that will function with the processor. We hope to remedy this shortly.

PREFACE

If you find a snarf in the book, please bring it to my attention. If it's serious enough, you may win a six-pack of Coors.

May 1977 Revision

Affecting pages:

PROCESSOR -1
-3
-4
-13*
-25
-28
-37
-38
-44*
-55
INSTRUCTIONS-17
ASSEMBLER -6
-7
APPENDIX -6
-8
-10

*Non-significant typing error only

June 1978 Revision

Affecting pages:

PROCESSOR -1
-4
-5
-6
-24*
-38*
-59

PREFACE

June 1978 Revision (Cont.)

INSTRUCTIONS	-3*
	-12
	-14
	-15
	-17
	-19
ASSEMBLER	-7
	-9
APPENDIX	-8
	-14
	-16
	-29
	-32
	-34
	-35

*Non-significant typing error only

Ed Miller
Ft. Collins CPD
June 1978

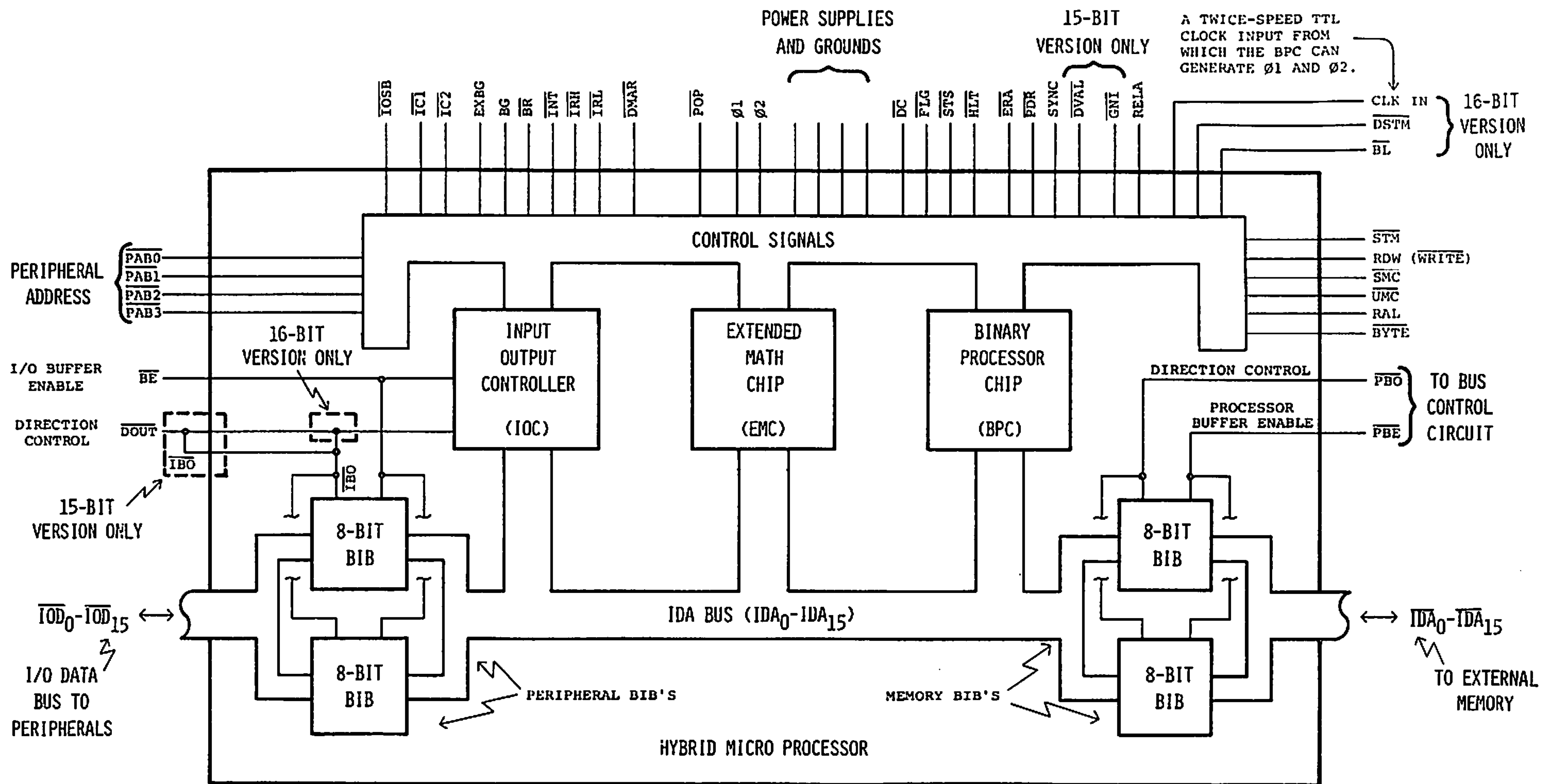


Figure P-1. Simplified Block Diagram of the Processor.

DESCRIPTION OF THE PROCESSOR

GENERAL INFORMATION

The CPD Processor consists of seven integrated circuits mounted on a ceramic substrate. Of these, three are N-channel MOS LSI chips. The remaining four chips are entirely bi-polar and serve as buffers to connect the LSI circuitry of the other chips to circuitry external to the substrate. Because the processor is an assemblage of components mounted on a substrate, it is often referred to as the "hybrid", "hybrid micro-processor", or simply as the "processor".

Figure P-1 is a simplified block diagram of the processor. The LSI chips are the Binary Processor Chip (BPC), Input-Output Controller (IOC), and the Extended Math Chip (EMC). All of the processing capability of the processor resides in those three chips; except for inversion the four Bi-Directional Interface Buffers (BIB's) are logically powerless. The three LSI chips communicate among themselves, and also with the outside world, via a collection of control signals and a 16-bit bus called the IDA Bus (IDA stands for Instruction/Data/Address).

The processor is available in two versions. One version uses 15-bit addressing for a maximum memory size of 32K words, and implements multi-level indirect addressing. The other version uses 16-bit addressing for a maximum memory size of 64K words, and implements a single level of indirect addressing. The 15-bit processor uses 15-bit versions of the BPC and IOC; the 16-bit processor uses 16-bit versions. The EMC is currently a 16-bit version that works in either processor; an obsolete 15-bit version of the EMC also exists but is not currently being produced.

The two versions of the processor are far more alike than they are different. Some new machine-instructions were added for the 16-bit IOC. However, they represent an alternate method of doing something (in light of the different way the 16th address bit is used) rather than a major extension of capability. Other than for size, both processors are alike in the general way they interface to memory. Their sets of machine-instructions are nearly identical; in fact, an assembler exists that can be used for both. The information in this book is generally applicable to both processors; information that applies to a particular version is labeled as such.

The IDA Bus is buffered as it leaves the hybrid, but the control signals are not. The BIB's are grouped together to buffer the IDA Bus in a way that allows it to perform two different functions. Each BIB can buffer eight bits of the IDA Bus. Two BIB's are grouped together to connect the IDA Bus to the (main and external) memory; those BIB's are called the Memory BIB's. The remaining two BIB's are grouped together to connect the IDA Bus to the IOD Bus. The IOD Bus (I/O Data Bus) is the data bus that serves peripheral devices. Accordingly, the BIB's connecting the IDA Bus with the IOD Bus are called the Peripheral BIB's. The Memory BIB's are enabled by a circuit (external to the hybrid) which detects memory traffic on the IDA Bus. The Peripheral BIB's are controlled by the IOC as the various types of input-output operations are performed.

GENERAL INFORMATION (CONT.)

Figure P-2 illustrates the nature of the BIB's. Each bit of the IDA Bus is buffered in both directions by tri-state buffers controlled by non-overlapping buffer enable signals.

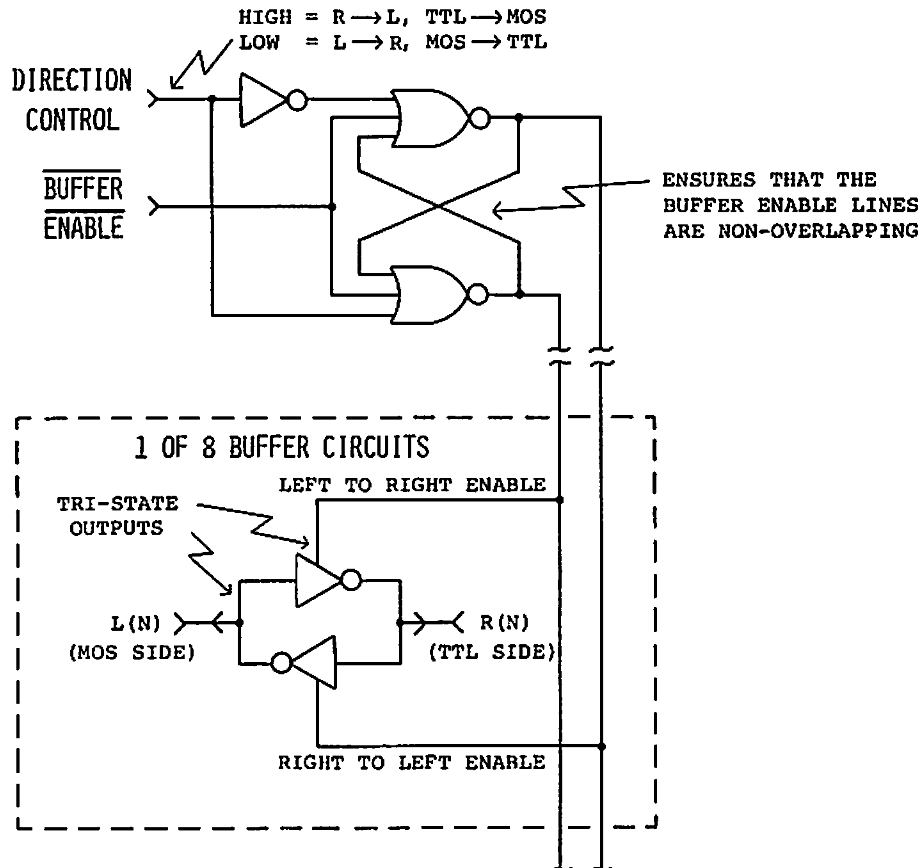


Figure P-2. Nature of the BIB's.

MEMORY CONVENTIONS

The term "memory" will be used to refer to any addressable memory location, regardless of whether that location is physically within the hybrid micro-processor, or external to it. The term "external memory" refers to memory that is not physically within the hybrid. The term "register" refers to various storage locations within the hybrid micro-processor itself. These registers range in size from 1 to 16 bits. Most of the registers are 16 bit registers. The term "addressable register" refers to a register within one of the LSI chips that responds as memory when addressed. Most registers are not addressable. In most of the discussions that follow the context clarifies whether or not a register is addressable so that it is deemed unnecessary to explicitly differentiate between addressable and non-addressable registers. Those registers that are addressable are included in the meaning of the term "memory". The term "memory cycle" refers to a read or write operation involving a memory location.

MEMORY CONVENTIONS (CONT.)

The first 32 memory addresses do not refer to external memory. Instead, these addresses ($0-37_8$) are reserved to designate addressable registers within the micro-processor. Table P-I lists the addressable registers within the micro-processor.

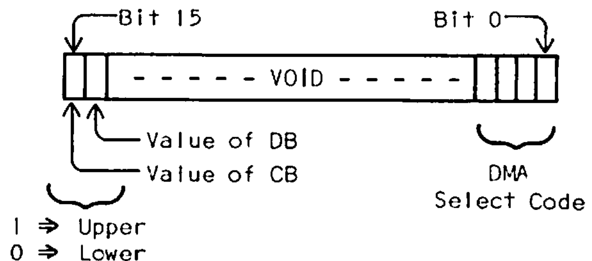
Table P-I. Addressable Registers.

	Octal Address	Name	Location	Description (# of Bits)
	0	A	BPC	Arithmetic Accumulator (16)
	1	B	BPC	Arithmetic Accumulator (16)
	2	P	BPC	Program Location Counter (least 15 of 16 or 16)
	3	R	BPC	Return Stack Pointer (least 15 of 16 or 16)
	4	R4	IOC	Peripheral Activity Designator (—)
	5	R5	IOC	Peripheral Activity Designation (—)
	6	R6	IOC	Peripheral Activity Designator (—)
	7	R7	IOC	Peripheral Activity Designator (—)
	10	IV	IOC	Interrupt Vector (upper 12 of 16)
* →	11	PA	IOC	Peripheral Address Register (least 4 of 16)
	12	W	IOC	Working Register (16)
† →	13	DMAPA	IOC	2 MSB = CB & DB; 4 LSB = DMA Periph. Add. Reg.
	14	DMAMA	IOC	DMA Memory Address & Direction Register (16)
	15	DMAC	IOC	DMA Count Register (16)
	16	C	IOC	Stack Pointer (16)
	17	D	IOC	Stack Pointer (16)
	20-23	AR2	EMC	BCD Arithmetic Accumulator (4 x 16)
	24	SE	EMC	Shift Extend Register (least 4 of 16)
* →	25-27	X	EMC	Internal Arithmetic Register (3 X 16)
	30-37	UNASSIGNED		
	77770/ 177770	AR1	R/W	BCD Arithmetic Register (4 x 16)

* Not available for general use. Part of processes internal to a chip. It is best to pretend that these registers do not exist.

† Read register 13_8 produces:

CB and DB are actually discrete registers, and while they can only be read by reading R13, storinging into R13 will not alter their values. Use the CBL, CBU, DBL and DBU machine instructions for that purpose. CB and DB exist in the 16-bit version only.



MEMORY CONVENTIONS (CONT.)

Most of the traffic on the IDA Bus has to do with memory. Both address of memory locations, and the contents of those locations (data and machine-instructions) are transmitted over the same 16-bit bi-directional bus (the IDA Bus). Further, memory can be physically distributed along the Bus. Each of the three chips in the processor contains registers which are addressable, and addressable memory also exists external to the processor.

MEMORY CYCLES

A memory cycle involves some control lines as well as the IDA Bus. Start Memory ($\overline{STM}$) is used to initiate a memory cycle by identifying the contents of the IDA Bus as an address. Memory Complete* is used to identify the conclusion of a memory cycle. A line called Read/Write (RDW) specifies the direction of data movement; out of or into memory, respectively.

Each element in the system decodes the addresses for which it contains addressable memory. To initiate a memory cycle, an element of the processor puts the address of the desired location on the IDA Bus, sets the Read/Write line, and gives Start Memory. Then, elsewhere in the system the address is decoded and recognized, and that agency begins to function as memory. It is part of the system definition that whatever is on the IDA Bus when a Start Memory is given is an address of a memory (or register) location.

Here is a complete description of the entire process: An originator originates a memory cycle by putting the address on the IDA Bus, setting the Read/Write line, and giving a Start Memory. The respondent identifies itself as containing the object location of the memory cycle, and handles the data. If the originator is a sender (write) it puts and holds the data on the IDA Bus until the respondent acknowledges receipt by sending Memory Complete. If the originator is a receiver (read) the respondent obtains and puts the data onto the IDA Bus and then sends Memory Complete. The originator then has one clock time to capture the data; no additional acknowledgement is involved.

Figures P-3 and P-4 illustrate typical memory cycles.

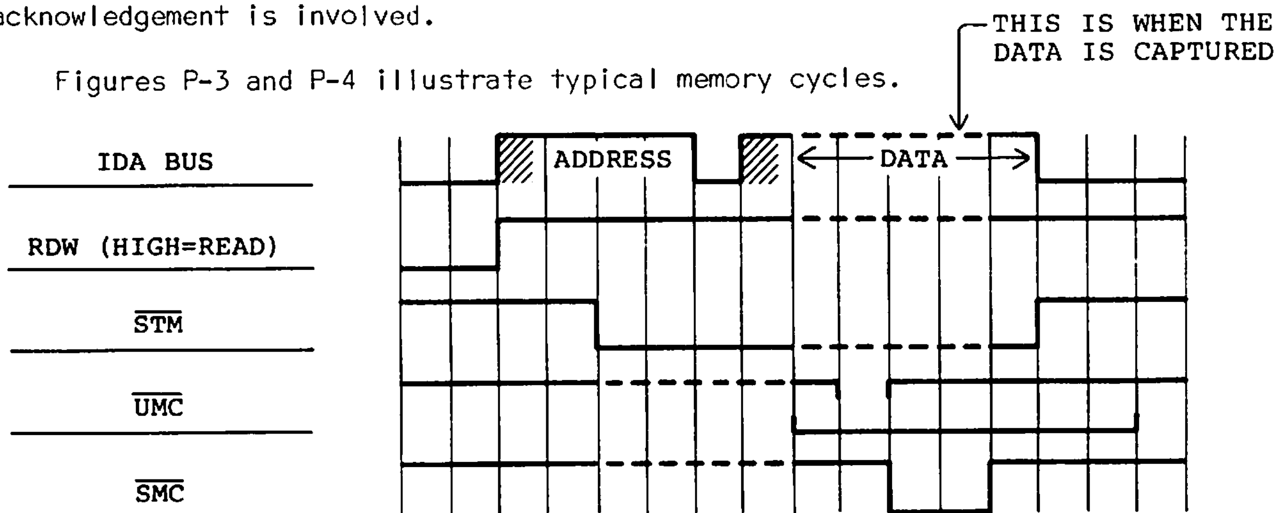


Figure P-3. Simplified Read Memory Cycle.

* There is no single signal called "Memory Complete". Instead there is Unsynchronized Memory Complete ($\overline{UMC}$) and Synchronized Memory Complete ($\overline{SMC}$). They mean the same thing for our present purposes, and their exact differences need not concern us here.

DESCRIPTION OF THE PROCESSOR

MEMORY CONVENTIONS

MEMORY CYCLES (CONT.)

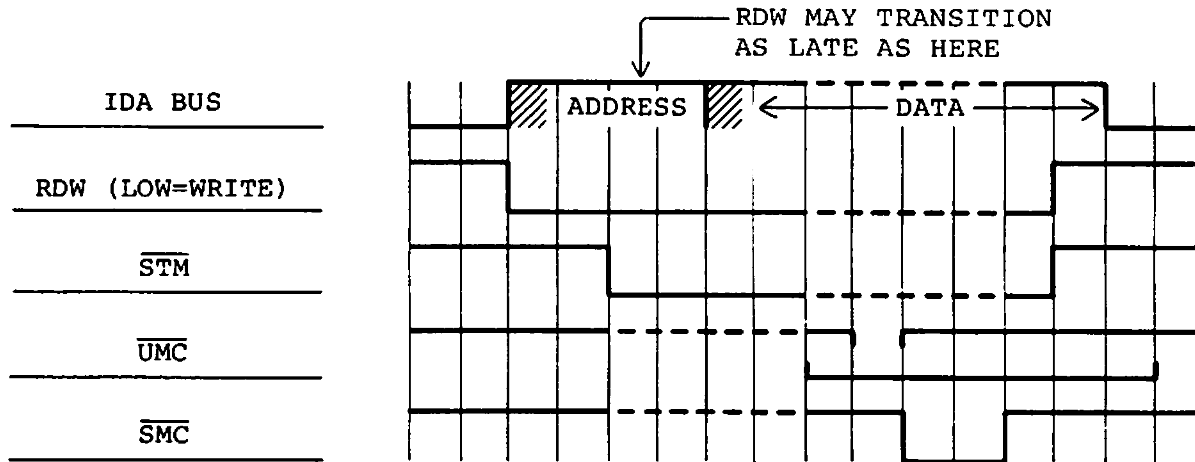


Figure P-4. Simplified Write Memory Cycle.

THE BYTE LINE

The IOC generates a signal called BYTE that affects memory operation. BYTE signifies that a memory cycle is to involve a left-half or right-half of a word rather than the entire word. The IOC is the only entity that is allowed to generate BYTE, which is used during the execution of certain IOC machine-instructions (the place and withdraw byte instructions).

During a read memory cycle the memory can supply the entire word regardless of the status of the BYTE line; the IOC will automatically extract the desired byte from the supplied word. However, during a write memory cycle the memory must merge the transmitted byte with the existing other half of the word (which is already in memory). The transmitted byte will be sent as the left-half or right-half of a word (that is, on the upper eight bits or on the lower eight bits of the IDA Bus), as is appropriate for whichever byte it is supposed to be.

The 15-bit and 16-bit versions of the IOC differ in the way they indicate which half of the word is being sent to memory. (These indicators are actually in force for both read and write memory cycles, but may be entirely ignored during read memory cycles.) For 15-bit IOC's the left-right information appears in the left-most bit of the address word; only 15 bits are needed for addressing the word anyway. In this scheme a one in bit 15 indicates a left-half. For 16-bit IOC's the entire 16 bits is required for addressing, and a separate signal (BL - Byte Left Not) is supplied to the memory. When bit 15 is used to designate the byte, bit 15 must be latched by the memory at the time the address is sent, as it is effectively sent as part of the address. On the other hand, BL is a steady state signal valid for the duration of the memory cycle.

When acting as memory themselves, none of the BPC, IOC, or EMC utilize the BYTE line during a write memory cycle. This means that a byte can be

MEMORY CONVENTIONS

THE BYTE LINE (CONT.)

read from a register in any of those chips, but that only entire words can be written to those registers.

RAL LINE

Among several service functions performed by the BPC for the IOC and EMC is the generation of a signal called RAL (Register Access Line) whenever an address on the IDA Bus is within the range reserved for register designation. RAL functions to prevent the external memory from responding to any memory cycle having such an address.

FUNCTIONAL DESCRIPTION OF THE BPC

The BPC has two main functions. The first is to fetch machine-instructions from memory for itself, the IOC, and for the EMC. A fetched instruction may pertain to one or more of those chips. A chip that is not associated with a fetched instruction simply ignores that instruction. The second main function of the BPC is to execute the 56 instructions in its own repertoire. These instructions include general purpose register and memory reference instructions, branching instructions, bit manipulation instructions, and some binary arithmetic instructions. Most of the BPC's instructions involve one of the two accumulator registers: A and B.

There are four addressable registers within the BPC and they have the following functions: The A and B registers are used as accumulator registers for arithmetic operations, and also as source or destination locations for most BPC machine-instructions referencing memory. The R register is an indirect pointer into an area of read/write memory designated to store return addresses associated with nests of subroutines encountered during program execution. The P register contains the program counter; its value is the address of the memory location from which the next machine-instruction will be fetched.

Upon the completion of each instruction the program counter (P register) has been incremented by one, except for the instructions JMP, JSM, RET, and SKIP instructions whose SKIP condition has been met. For those instructions the value of P will depend on the activity of the particular instruction.

INDIRECT ADDRESSING

Indirect addressing is a technique in which an instruction that references memory treats the first one or more references as intermediate steps to referencing the final destination. Each intermediate reference yields the address of the next location to be referenced. When an intermediate location can point to yet another intermediate location, such addressing is termed *multi-level* indirect addressing. Indirect addressing is not a property of the memory; it is property of the chips that use the memory. Any chip that is to implement instructions employing indirect addressing must contain a special gear works for that purpose.

MULTI-LEVEL INDIRECT ADDRESSING

BPC's that can address 32K of memory can perform multi-level indirect addressing. Memory addresses appear on the IDA Bus as 15-bit patterns during the address portion of a memory cycle. The BPC machine-instructions that reference memory are capable of multi-level indirect addressing. The initial indirect indicator is a particular bit in the machine-instruction itself (the most-significant, or left-most, bit: Bit 15). The internal operation of the BPC is so arranged that if the memory content of that address also has a one in bit 15, the other bits of the contents are themselves taken as an indirect address. The process of accessing via an indirect address continues

INDIRECT ADDRESSING

MULTI-LEVEL INDIRECT ADDRESSING (CONT.)

until a location is accessed which does not have a one in bit 15. At that time the content of that location is taken as the final address; that is, it is taken to be the address of the desired location and the memory cycle is completed when that final desired location is accessed.

SINGLE LEVEL INDIRECT ADDRESSING

BPC's that can address 64K of memory are not capable of multi-level indirect addressing; they can perform only one level of indirect addressing. As before, bit 15 of the particular memory reference instruction will be set. The contents of the referenced location will be read, and its entire 16-bit contents treated as the address of the final destination to be read from or written into. This is because addressing 64K of memory requires the use of bit 15 as an actual address bit; thus bit 15 is not available to indicate that the remaining bits are an indirect address. The format of the memory reference instructions themselves has not changed; bit 15 of those instructions still indicates an initial indirect reference, but no further indirect references can be indicated as memory is read. Hence only one level of indirect addressing is possible.

MEMORY REFERENCE INSTRUCTIONS & PAGE ADDRESSING

Machine-instructions fetched from memory are 16-bit instructions. Some of those bits represent the particular type to which the particular instruction belongs. Other bits differentiate the instruction from others of the same type. If a BPC machine-instruction is one that involves reading from, storing into, or otherwise manipulating the contents of a memory location, it is said to be a *memory reference instruction.* Load into A (LDA), Store from B (STB), and Jump (JMP) are examples. There are 14 memory reference instructions and they each contain bits to represent the address of the location that is to be referenced by the instruction. Only ten bits are devoted to indicating the address to be referenced. Those ten bits represent one of 1024_{10} locations on either the *base page* or the *current page* of memory. An additional bit in the machine-instruction indicates which. The base page is always a particular, non-changing, range of addresses, exactly 1024_{10} in number. A memory reference machine-instruction fetched from any location in memory (i.e., from any value of the program counter) may directly reference (that is, need not use indirect addressing) any location on the base page.

For 15-bit addressing the base page is addresses 00000_8-00777_8 and 77000_8-77777_8. For 16-bit addressing the base page addresses are 000000_8-000777_8 and 177000_8-177777_8. Figure P-5 depicts the base page.

There are two types of current pages. Each type is also 1024_{10} consecutive words in length. Except for base page references, a memory reference machine-instruction can directly reference only locations that are on the same current page as it; that is, locations that are within the page containing the current value of the

MEMORY REFERENCE INSTRUCTIONS & PAGE ADDRESSING (CONT.)

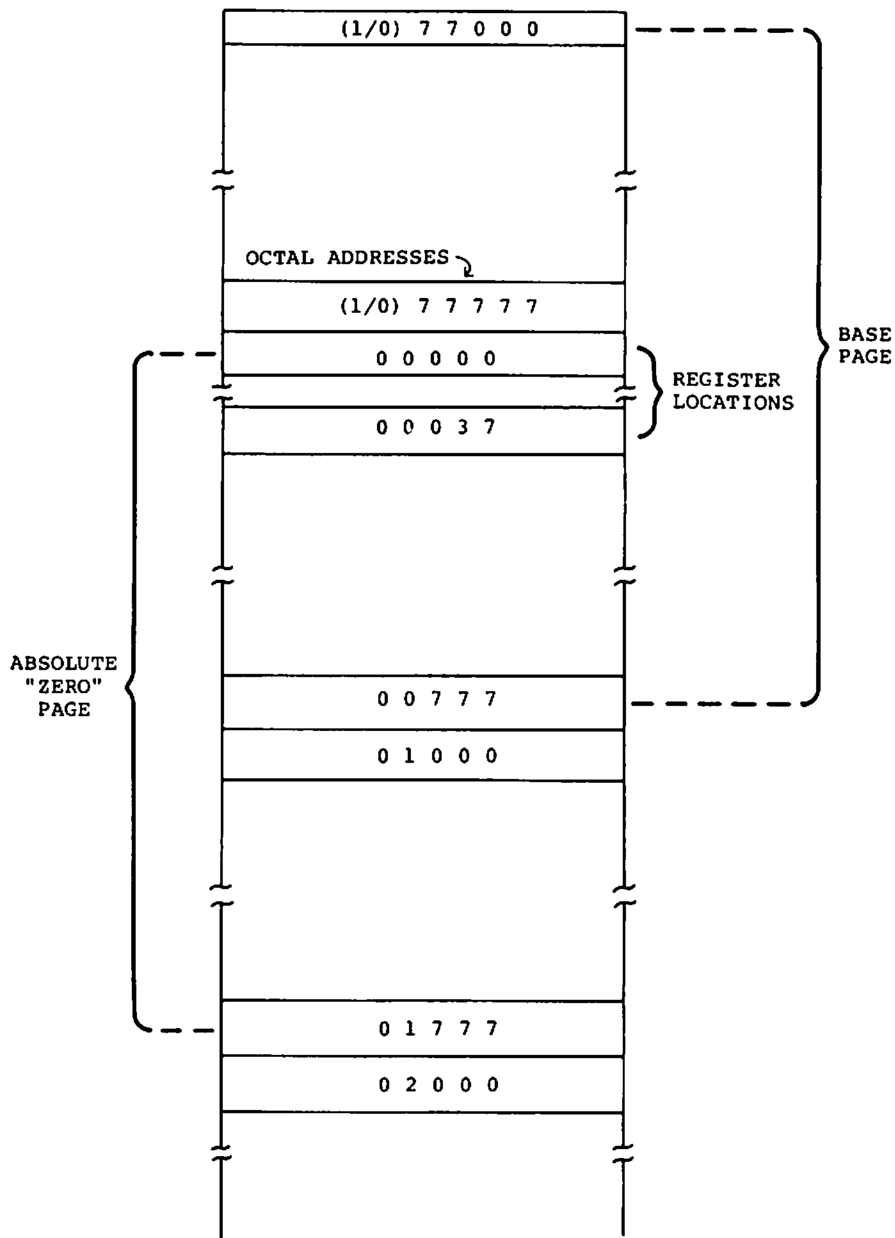


Figure P-5. Base Page Description.

program counter (P).* Thus the value of P determines the particular collection of addresses that are the current page at any given time. This is done in one of two distinct ways, and the particular way is determined by whether the signal called RELA is grounded or not. If RELA is ungrounded, the BPC is said to address memory in the "relative" mode. If RELA is grounded it is said to operate in the "absolute" mode.

* Off-page references that are not base page references must be made using indirect addressing.

FUNCTIONAL DESCRIPTION OF THE BPC

MEMORY REFERENCE INSTRUCTIONS & PAGE ADDRESSING (CONT.)

During the execution of each memory reference machine-instruction the BPC forms a full 15-bit or 16-bit address based on the ten bits of address contained within the instruction. How the supplied ten bits are manipulated before becoming part of the actual address, and how the remaining five or six bits are supplied, depends upon whether the instruction calls for a base page reference or not, and upon whether the addressing mode is relative or absolute. The differences are determined primarily by the two different definitions of the current page; one for each mode of addressing. Base page addressing is the same in either mode.

ABSOLUTE ADDRESSING

In the absolute mode of addressing the memory address space is divided into a base page and 32 or 64 possible current pages. The possible current pages are the consecutive 1024_{10} word groups beginning with 00000_8. The possible current pages can be numbered, 0 through 31_{10}; or 0 through 63_{10}. Thus the "zero page" is addresses 00000_8-01777_8. Note that the base page is not the same as the zero page; the base page overlaps pages zero and 31 for 32K machines, and overlaps pages zero and 63 for 64K machines.

RELATIVE ADDRESSING

In relative addressing there are as many possible current pages as there are values of the program counter. In the relative addressing mode a current page is the 512_{10} consecutive locations prior (that is, having lower valued addresses) to the current location (value of P), and the 511_{10} consecutive locations following the current location.

BASE PAGE ADDRESSING

All memory reference machine-instructions include a 10-bit field that specifies the location referenced by the instruction. What goes in this field is a displacement from some reference location, as an actual complete address has too many bits in it to fit in the instruction. This 10-bit field is bit 0 through bit 9. Bit 10 tells whether the referenced location is on the base page, or someplace else. Bit 10 is called the B/C bit, as it alone is used to indicate base page references. Bit 10 will be a zero if the reference is to the base page, and a one if otherwise.

If bit 10 is zero for a memory reference instruction (base page reference), the 10-bit field is sufficient to indicate completely which of the 1024 locations on the base page is to be referenced. There are two way to describe the rule that is the correspondence between bit patterns in the 10-bit field, and the locations that are the base page: (1) the least significant 10 bits of the "real address" (i.e., $(1)77{,}000_8$ through 777_8) are put into the 10-bit field, bit for bit; (2) as a displacement between $+777_8$ and -1000_8 about 0, with bit 9 being the sign.

The 32 register addresses are considered to be a part of the base page. Base page addressing is always done in the manner indicated above, regardless of whether relative or non-relative addressing is employed by the BPC.

MEMORY REFERENCE INSTRUCTIONS & PAGE ADDRESSING (CONT.)

CURRENT PAGE ADDRESSING

Current page addressing refers to memory reference instructions which reference a location which is not on the base page. The same 10-bit field of the machine-instruction is involved, but the B/C bit is a one (C). Now, since there are more than 1024 locations that are not the base page, the 10-bit field by itself, is not enough to completely specify the exact location involved. An assumption has to be made about which page of the memory is involved.

For absolute addressing the assumption is that the most significant 5 (or 6) bits of the P register correspond to the page, and the last 10 bits of the machine-instruction determine the location within that page. This assumption requires that there will be no page changes except by certain ways. This means that once the program counter is set to a particular location the top 5 (or 6) bits need not be changed for any addressing on that (which ever it is) page. When the assembler assembles a memory reference instruction, it computes the least 10 bits and puts them in the instruction. When the BPC executes the instruction it concatenates its own top 5 (or 6) bits of P with the address represented by the least 10 bits of the instruction; that produces the complete address for the location referenced by the instruction.

However, the least 10 bits produces by the assembler and placed in the machine-instruction do not correspond exactly to the "real" memory address that is referenced. Bit 9 (the 10th bit) is complemented before it is placed in the address field of the instruction. The other 9 bits are left unchanged. This induces a one-half page offset whose effect is to make current page addressing relative to the middle of the page. Table P-2 depicts current page absolute addressing. This similarity between current page and base page addressing is deliberate, and results in simplified hardware in the BPC.

Page changes can be accomplished in two ways: incrementing or decrementing the program counter in the BPC, and through indirect addressing. An example of incrementing to a new page is a continuous block of code that spans two adjacent pages. A page change through an increment or decrement can occur in the same general way due to skip instructions.

Indirect addressing allows page changes because the object of an indirect reference is always taken as a full 15-bit or 16-bit address. Indirect addressing is the method used for an instruction on a given page to either reference a memory location on another page (LDA, STA, etc.), or, to jump (JMP or JSM) to a location on another page.

Instructions on any page can make references to any location on the base page without using indirect addressing. This is because the B/C bit designates whether the 10-bit field in the instruction refers to the base page or to the current page. If B/C is a zero (B), the BPC automatically assumes the upper 5 or 6 bits are all zeros, and thus the 10-bit field refers to the base page. If B/C is a one (C), the top 5 or 6 bits are taken for what they are, and the current page is referenced (whichever it is).

MEMORY REFERENCE INSTRUCTIONS & PAGE ADDRESSING

CURRENT PAGE ADDRESSING (CONT.)

Table P-2. Current Page Absolute Addressing for Memory Reference Instructions.

LEAST 10 BITS OF ASSEMBLER OUTPUT (octal)	"REAL OCTAL ADDRESS"	
	TOP 5 (6) BITS OF P	LOWER 10 BITS
1 0 0 0	X X START OF PAGE	0 0 0 0
1 0 0 1	X X	0 0 0 1
1 0 0 2	X X	0 0 0 2
.	.	.
.	.	.
.	.	.
1 7 7 7	.	0 7 7 7
0 0 0 0	.	1 0 0 0
0 0 0 1	.	1 0 0 1
0 0 0 2	.	1 0 0 2
.	.	.
.	.	.
.	.	.
0 7 7 7	X X END OF PAGE	1 7 7 7

It is the responsibility of the assembler to control the B/C bit at the time the machine-instruction is assembled. It does this easily enough by determining if the address of the operand (or its "value") of an instruction is in the range of (1)77,000$_8$ through 0, or, 0 through 777$_8$. If it is, then it is a base page reference and B/C is made a zero for that instruction.

Relative addressing does not require the concept of a fixed page, as in absolute addressing. The word "page" can still be used, but requires a new definition:

MEMORY REFERENCE INSTRUCTIONS & PAGE ADDRESSING

CURRENT PAGE ADDRESSING (CONT.)

In relative addressing, a page is 1024_{10} consecutive locations, having 512_{10} locations prior to the current location, and 511_{10} locations following the current location.

As before, direct addressing is possible anywhere within the page. But off-page references (other than to the base page) require indirect addressing, which, once started, works as before - it is not relative, but produces a full 15-bit or 16-bit absolute address.

Figure P-6 illustrates relative addressing. Relative current page addressing is done much the same way as base page addressing. The 10-bit field in the memory reference instructions is encoded with a displacement relative to the current location.

Bit 9 (the 10th, and most significant bit of the 10) is a sign bit. If it is a zero, then the displacement is positive, and bits 0-8 are taken at face value. If bit 9 is a one, the displacement is negative. Bits 0-8 have been complemented and then incremented (two's complement) before being placed in the field. To get the absolute value of the displacement, simply complement them again, and increment, ignoring bit 9.

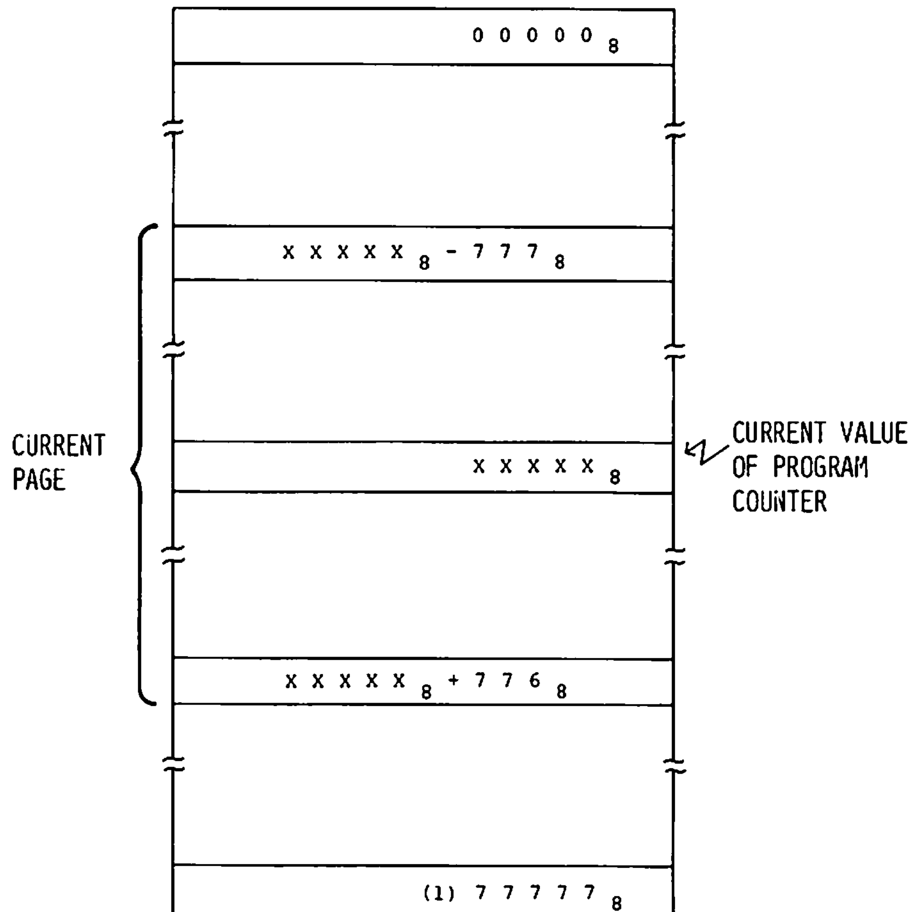


Figure P-6. Relative Addressing.

FUNCTIONAL DESCRIPTION OF THE BPC

SUBROUTINES

The processor implements subroutines in the following way. The JSM memory reference instruction is used to cause a jump (change in value of P) to the start of the subroutine. Also as part of the JSM, the BPC saves the value of P that corresponds to the word of programming that is the JSM. That value is saved in a section of read/write memory called the *return stack*.

The return stack is a group of contiguous locations, whose starting address less one was initially stored in the R register (in the BPC). Thus R is an indirect pointer. What a JSM does is to increment the value in R and then use that new value as the address at which to store the value of P that is to be saved. Once this activity is complete, P is actually set to the address of the first word of the subroutine and its execution commences.

A subroutine is terminated with a RET n instruction. The essence of this instruction is to read the location that R points at, set P to that value plus n, and then decrement R. The garden variety return is a RET 1. Different values of n permit different returns corresponding to error or other special conditions.

Subroutines can be nested as deep as the size of the return stack will allow. The subroutines themselves can be either in ROM or read/write memory.

FLAGS

The BPC is capable of branching based on the condition of each of four signals externally supplied to the chip. These signals are Decimal Carry (DC), Halt (HLT), Flag (FLG), and Status (STS). The EMC acts as a source for Decimal Carry, which represents an overflow condition during certain arithmetic operations performed by the EMC. The other signals can be defined in any way that is suitable for the system in which the processor is operating; they are not used for inter-chip communication within the processor.

BUS REQUESTS AND INTERRUPTS

Two protocols that do involve inter-chip communication are those of Bus Request and Interrupt. Bus Request ($\overline{BR}$) provides a way for a chip in the processor, or even a device external to the processor, to request unfettered use of the IDA Bus. A signal called Bus Grant (BG) is generated if all chips and any other interested entities agree to do so. The requesting agency can use the IDA Bus for whatever purpose it wants, (typically to do memory cycles). During the time that Bus Grant is in effect all chips suspend their activity. Bus Grants can be given even in the middle of the execution of an instruction. Because of this, the chips do not grant bus requests indiscriminately. Furthermore, a Bus Grant not requested by the IOC is used by the IOC to create *Extended Bus Grant* (EXBG), which is routed from chip to chip in a definite order; chips or other entities not at the top of the chain can exercise the right not to pass along the signal. This

BUS REQUESTS AND INTERRUPTS (CONT.)

allows a Bus Request from the IOC to have a higher priority than any entity further down the chain. Even if both are requesting the bus, the IOC can "steal" EXBG by not passing it along. Further down the chain from the IOC, BG serves to indicate only that the bus is being granted to somebody; a paticular requesting device must wait until it sees EXBG before it can use the bus.

The Bus Request protocol includes these additional considerations: Any entity on the bus may ground BG as long as BG is not already being given. This allows any entity anywhere in the chain to protect its own access to the bus against all agencies. Further, the BPC itself refuses to issue a BG as long as any memory cycle is in progress.

Figure P-7 illustrates the usage of the Bus Request, Bus Grant, and Extended Bus Grant protocol.

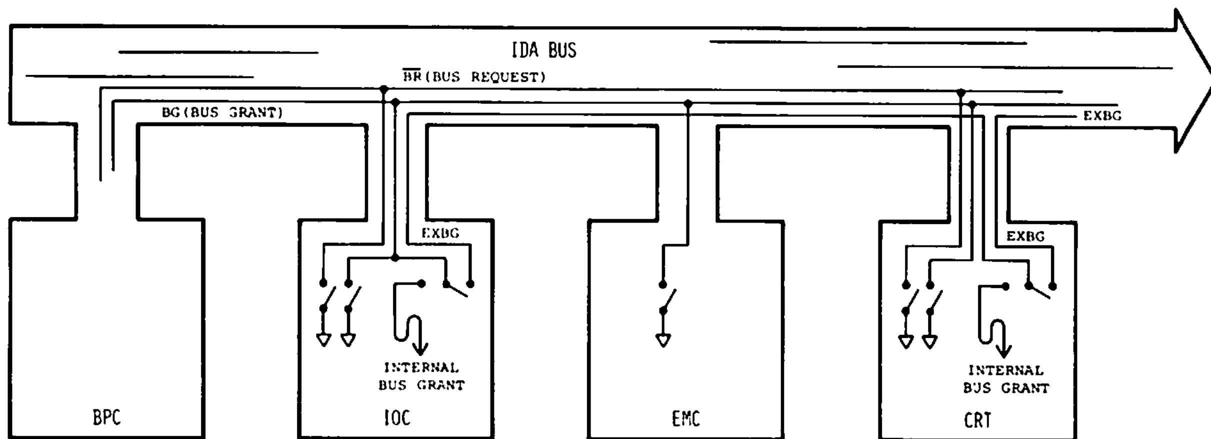


Figure P-7. Bus Request Protocol.

Following is a description of how the inter-chip mechanism for interrupt acts. During an instruction fetch a line called Interrupt ($\overline{INT}$) can signal that the IOC has agreed to allow an interrupt requested by a peripheral. The management of this decision is complicated and its description belongs with a description of the IOC. However, once the decision is made, the IOC signals the BPC with $\overline{INT}$. This has to occur during a certain period of time ending with the end of the instruction fetch. (A signal called SYNC indentifies the instruction fetch.)

What the chips in the system must do when an interrupt occurs is to abort the execution of the instruction just fetched (it will be fetched again, later). Then the BPC executes the instruction JSM 10_8 Indirect.

BUS REQUESTS AND INTERRUPTS (CONT.)

Register address 10_8 is located in the IOC, and is the Interrupt Vector register (IV). That register is a pointer into a stack of addresses of the starting locations for the various interrupt service routines. These routines handle the traffic needed by the interrupting peripheral. A special mechanism in the IOC sets the bottom four bits of IV to correspond to the particular peripheral that requested the interrupt. Thus IV points to different service routines, according to which peripheral interrupted.

In any event, the JSM 10_8 Indirect causes the value of P for the aborted instruction to be saved on the return stack. A RET 0 at the end of the service routine results in that very instruction being fetched over again, at the conclusion of the service routine.

FUNCTIONAL DESCRIPTION OF THE IOC

The IOC has two main functions. One is to manage the transfer of information between the processor and external peripheral devices. This is done by providing capabilities classified as Standard I/O, Interrupt and Direct Memory Access (DMA). The second main function is to provide machine-instructions allowing software management of two stacks in Read/Write Memory.

To implement these tasks the IOC contains a number of addressable registers. The function of each will be discussed as the various topics of IOC operation are covered.

GENERAL INFORMATION ABOUT I/O

The IOC allows up to 16 peripheral devices to be present at one time. Each peripheral device is connected to the IOD Bus, Peripheral Address Bus, and the various control signals necessary for that particular device's operation. Individual I/O operations (exchanges of single words) occur between the processor and one peripheral at a time, although Interrupt and DMA modes of operation can cause automatic interleaving of individual operations. A select code transmitted by the Peripheral Address Bus ($\overline{PAB0}$-$\overline{PAB3}$) indentifies which of the 16 devices is the object of an individual I/O operation.

In addition, the peripheral interface is the source of the Flag and Status bits for the BPC instructions SFS, SFC, SSS, and SSC. Since there can be many interfaces, but only one each of Flag and Status, only the interface addressed by the select code is allowed to ground these lines. Their logic is such that if the addressed peripheral is not present on the I/O Bus, Status and Flag are logically false.

$\overline{IC1}$ and $\overline{IC2}$ are two control lines that are sent to each peripheral interface by the IOC. The state of these two lines during the non-DMA transfer of information can be decoded to mean something by the interface. Just what 'something' will be is subject to agreement between the firmware designer and the interface designer - it can be anything they want, and might not be the same for different interfaces. These two lines act as a four position mode switch on the interface, controlled by the IOC during an I/O operation.

I/O BUS CYCLES

There are no specific machine-instructions for which the IOC responds by doing I/O operations. That is, there is no "output instruction", and no "input instruction". The real workhorse of I/O is a thing called an *I/O Bus Cycle*. An I/O Bus Cycle is an exchange of a word between the IDA Bus and the IOD Bus, via the Peripheral BIB's. The exchange is not of the handshake variety. I/O Bus Cycles are termed read or write I/O Bus Cycles, depending upon whether information is being read from, or written to, a peripheral.

Each of the three modes of I/O operation (Standard I/O, Interrupt, and

I/O BUS CYCLES (CONT.)

DMA) utilize I/O Bus Cycles. After we have examined how an I/O Bus Cycle works, the explanation of the various modes of I/O will amount to showing different ways to initiate I/O Bus Cycles.

For example, during Standard I/O operation, an I/O Bus Cycle is initiated by a reference to one of R4 through R7 in the IOC. One way that can be done is with a BPC memory reference instruction; for instance, STA R4 (for a write cycle), or LDA R4 (for a read cycle).

The IOC includes a register called the Peripheral Address Register (PA) which is used in establishing the select code surrently in use. The peripheral address is established by storing the desired select code into PA with an ordinary memory reference instruction. The bottom four bits of this register are brought out of the IOC as $\overline{PAB0}$ through $\overline{PAB3}$. Each peripheral interface decodes $\overline{PAB0}$-$\overline{PAB3}$ and thus determines if it is the addressed interface.

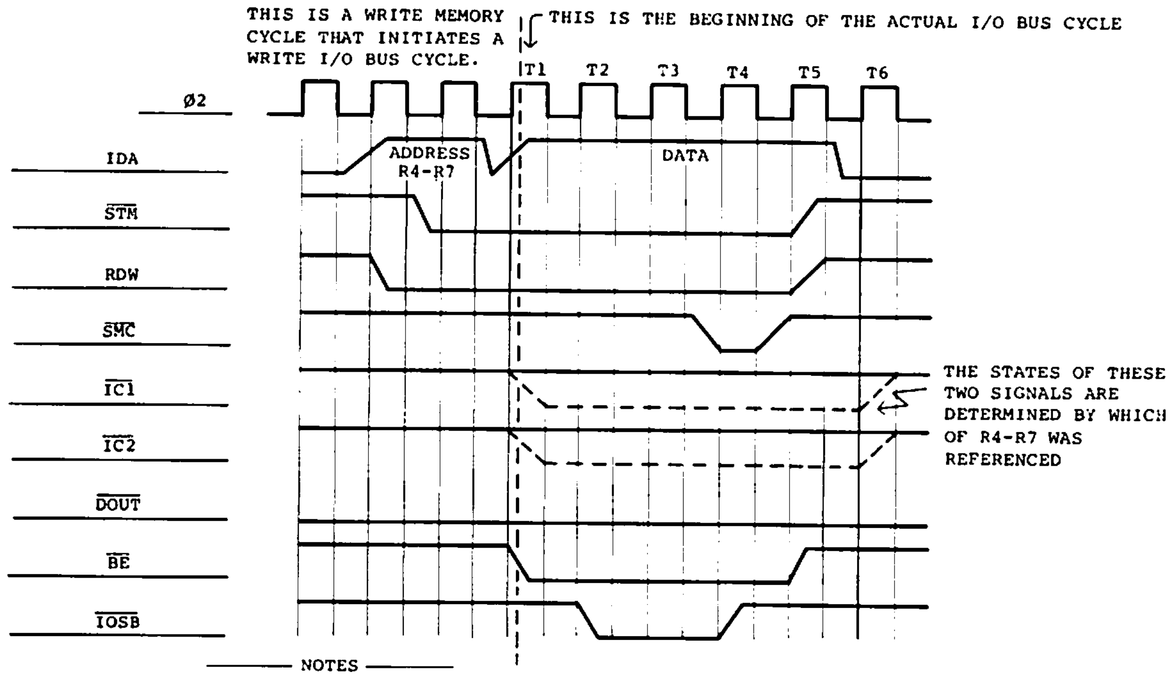
Consider a write I/O Bus Cycle as illustrated in Figure P-8. This is initiated with a reference to one of R4-R7. The IOC sees this as an address between 4 and 7 on the IDA Bus while $\overline{STM}$ is low. The Read line is low to denote a write operation. The IOC enables the Peripheral BIB's and specifies the direction. It also sets the control lines $\overline{IC1}$ and $\overline{IC2}$, according to which of R4 through R7 was referenced. Meanwhile, the BPC has put the word that is to be written onto the IDA Bus. Because both the Memory BIB's and Peripheral BIB's are enabled, that word is felt at all peripheral interfaces. The interface that is addressed uses $\overline{DOUT}$ to understand it's to read something, and uses $\overline{IOSB}$ as a strobe for doing it. After $\overline{IOSB}$ is given, the IOC gives [Synchronized] Memory Complete ($\overline{SMC}$) and the process terminates. The BPC has written a word to the interface whose select code matched the number in the PA register.

A read I/O Bus Cycle is similar, as shown in Figure P-9. Here the BPC expects to receive a word from the addressed peripheral interface. Read, $\overline{DOUT}$ and $\overline{BE}$ are different because the data is now moving in the other direction.

In either case, the critical control signals $\overline{SMC}$ and $\overline{IOSB}$ are given by the IOC, and their timing is fixed. There can be no delays due to something's not being ready, nor is there any handshake between the interface and the IOC.

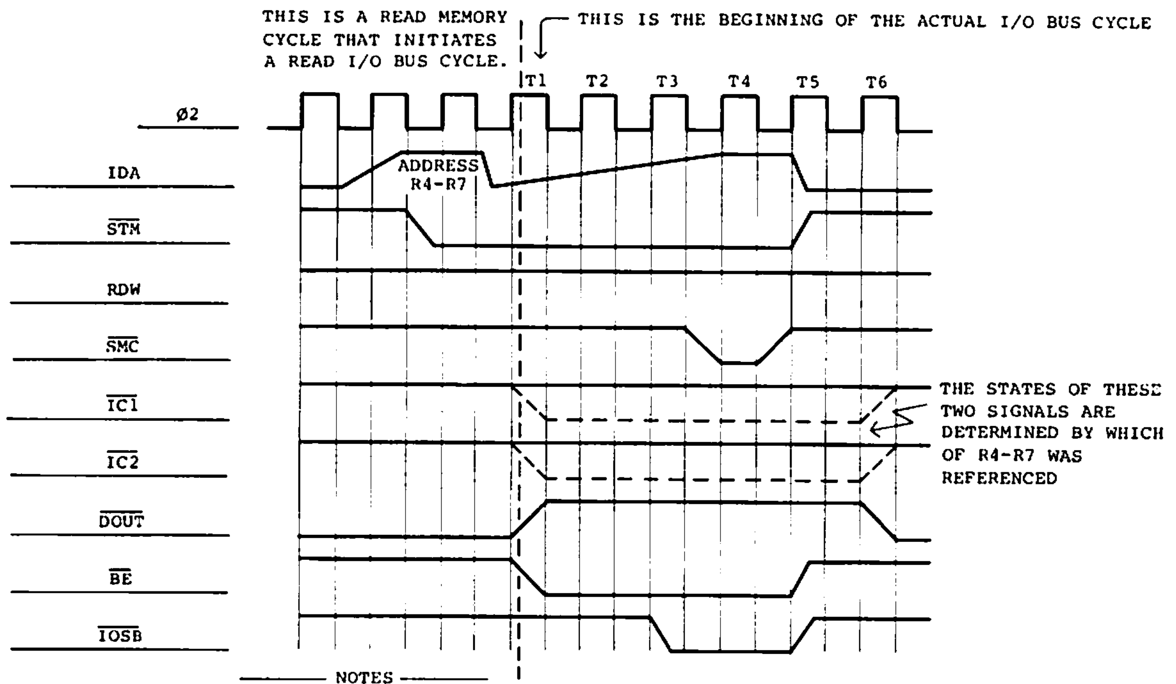
It is the responsibility of the firmware not to initiate an I/O Bus cycle involving a device that is not ready. To do so will result in lost data, and there will be no warning that this has happened.

I/O BUS CYCLES (CONT.)



1. THIS I/O BUS CYCLE WAS INITIATED BY ANY WRITE-INTO-MEMORY INSTRUCTION WHICH REFERENCED ONE OF R4 THRU R7.
2. CONTROL INFORMATION IS VALID ON BOTH EDGES OF IOSB.
3. DATA IS LATCHED INTO THE INTERFACE ON THE TRAILING EDGE OF IOSB.

Figure P-8. A Write I/O Bus Cycle.



1. THIS I/O BUS CYCLE WAS INITIATED BY ANY READ-FROM-MEMORY INSTRUCTION WHICH REFERENCED ONE OF R4 THRU R7.
2. CONTROL INFORMATION IS VALID ON BOTH EDGES OF IOSB.
3. DATA FROM THE INTERFACE IS LATCHED INTO THE BPC DURING T4.

Figure P-9. A Read I/O Bus Cycle.

FUNCTIONAL DESCRIPTION OF THE IOC

STANDARD I/O

Standard (programmed) I/O involves three activities:

1) Setting the peripheral address
2) Investigating the status of the peripheral
3) Initiating an I/O Bus Cycle

ADDRESSING THE PERIPHERAL

A peripheral is selected as the addressed peripheral by storing its octal select code into a 4-bit register called PA (Peripheral Address - address 11_8). Only the four least significant bits are used to represent the select code.

CHECKING STATUS

The addressed peripheral is allowed to control the Flag and Status lines. (That is, it is up to the interface to not ground Flag or Status unless it is the addressed interface.) These lines have an electrical logic such that when floating they appear false (clear, or not set) for SFS, SFC, SSS, and SSC.

The basic idea (and it can be done in a variety of ways) is to use sufficient checks of Flag and Status before and amongst the I/O Bus Cycles such that there is no possibility of initiating an I/O Bus Cycle to a device that is not ready to handle it. One way to do this with Standard I/O is to precede every I/O Bus Cycle with the appropriate checks.

INITIATING I/O BUS CYCLES

An I/O Bus Cycle occurs once each time one of R4 - R7 (4_8-7_8) is accessed as memory. An instruction that "puts" something into R4-R7 results in an output (write) I/O Bus Cycle. Conversely, an instruction that "gets" something from R4-R7 results in an input (read) I/O Bus Cycle. However, there are no R4 through R7. The use of address 4-7 is just a device to get an I/O Bus Cycle started; they do not correspond to actual physical registers in the IOC.

Consider the following hypothetical case, (specially invented for purposes of illustration) - Suppose we are to write a driver for a smarter than average paper tape punch: Upon a single command it can output 50 feed-frames for leader. The routine is to have two entry points; one for outputting a single word of data, and one for causing the leader. Also, the punch sets the status line if it gets low on tape. Prior to calling our driver, the main program puts the word to be outputted into DATA, and the select code of the punch in PUNSC.

```
 1.   PUNCH  JSM  SETUP   SET SELECT CODE, CHECK AVAILABILITY
 2.          LDA  DATA    GET OUTPUT DATA WORD
 3.          STA  R4      OUTPUT THE DATA (IC1 = 0, IC2 = 0)
 4.          RET  1       RETURN TO MAIN PROGRAM
 5.   LEADR  JSM  SETUP   SET SELECT CODE, CHECK AVAILABILITY
 6.          STB  R5      OUTPUT LEADER (IC1 = 1, IC2 = 0)
 7.          RET  1       RETURN TO MAIN PROGRAM
 8.   SETUP  LDA  PUNSC   GET SELECT CODE
 9.          STA  PA      PUT IT INOT PERIPHERAL ADDRESS REG
10.          SFC  *       WAIT IF PUNCH NOT AVAILABLE
```

STANDARD I/O

INITIATING I/O BUS CYCLES (CONT.)

```
11.           SSS    BXCRS    SKIP IF PUNCH OUT OF TAPE
12.           RET    1        OK, DO OUTPUT OPERATION
13.   BXCRS   :               HANDLE THE OUT OF TAPE SITUATION
              :
              :
14.   PUNSC   NOP             TAPE PUNCH SELECT CODE
15.   DATA    NOP             OUTPUT DATA WORD
```

Lines 1 and 5 invoke lines 8 through 12. Lines 8 and 9 set the select code, and line 10 checks for presence and availability (both must be "yes", or, at the interface the Flag will be false). Line 11 checks for the out-of-tape condition; it is the responsibility of the punch-interface combination to set Status high when the tape supply is low and the punch is addressed by PA. The routine at BXCRS handles the out of tape condition.

Lines 2 and 3 punch a word of data onto the tape. Line 3 causes a "write" (output) I/O Bus Cycle. The contents of (in this case) A are written to the addressed peripheral. Because it is R4 that is referenced, IC1 and IC2 are both zeros. The interface understands an output I/O Bus Cycle with IC1 and IC2 both zeros to be a command to punch the supplied word.

Line 6 gives the command to punch leader. Because it is a write operation referencing R5, an output I/O Bus Cycle is done with IC1 = 1 and IC2 = 0. In this instance the contents of B is sent to the punch (we will assume that it is ignored, however). The interface understands an output I/O Bus Cycle with IC1 = 1 and IC2 = 0 as the command to generate leader.

The 16-bit word transmitted from B need not be ignored. An even smarter punch might use it as the number of feed-frames to punch. A more general approach would be for the interface to recognize that IC1 = 1 and IC2 = 0 signifies that the accompanying word is to be decoded to determine the instruction/control information. The possibilities are numerous.

THE ODDBALL POSSIBILITIES

By this time in your reading you no doubt instantly recognize LDB R4 as an input operation where a word is read from the addressed peripheral and placed into B. But what about the other memory reference instructions? What, for instance, does ADA R4 do, or a CPA R4, or an ISZ R4, or worse still, a LDB R4,I? Some of these things do not have a known practical use, but they each work in a logically straight-forward manner.

An ADA R4 will read a word of data from the addressed peripheral, and then add it to the contents of A, leaving the result in A.

A CPA R4 will read a word of data from the addressed peripheral, and then compare that with the existing contents of A. The BPC will skip the next instruction if the two are unequal.

FUNCTIONAL DESCRIPTION OF THE IOC

STANDARD I/O

THE ODDBALL POSSIBILITIES (CONT.)

An ISZ R4 is an input/increment-and-skip/output instruction. It reads a word of data from the addressed peripheral and increments the resulting value. If the sum is zero, the next instruction is skipped. But in any case, the incremented value is written back to the same peripheral it came from. The interface sees a read I/O Bus Cycle followed a very short time later by a write I/O Bus Cycle.

An LDB R4,I does the obvious thing. A word of data is read from the addressed peripheral. Once the data is read it is treated exactly as if it had come from regular memory, and the action proceeds just as for any other Load B-indirect.

THE INTERRUPT SYSTEM

The idea behind interrupt is that for certain kinds of peripheral activity, the processor can go about other business once the I/O activity is intiated, leaving the bulk of the I/O activity to an interrupt service routine. When the peripheral is ready to handle another ration of data (it might be a single byte or a whole string of words) it requests an interrupt. When the processor grants the interrupt, the program segment currently being executed is automatically suspended, and there is an automatic JSM to an interrupt service routine that corresponds to the device that interrupted. The service routine uses Standard I/O to accomplish its task. A RET 0,P terminates the activity of the service routine and causes resumption of the suspended program.

PRIORITY

The interrupt system allows even an interrupt service routine to be interrupted and is therefore a multi-level interrupt system, and it has a priority scheme to determine whether to grant or ignore an interrupt request.

The IOC allows two levels of interrupt, and has an accompanying two levels of priority. Priority is determined by select code; select codes $0\text{-}7_8$ are the lower level (priority level 1), and select codes $10_8\text{-}17_8$ are the higher level (priority level 2). Level 2 devices have priority over level 1 devices; that is, a disc drive operating at level 2 could interrupt a plotter operating at level 1, but not vice versa. Within a priority level all devices are of "equal" priority, and operation is of a first come-first served basis; a level 1 device cannot be interrupted by another level 1 device, but only by a level 2 device. However, priorities are not equal in the case of simultaneous requests by two or more devices on the same level. In such an instance the device with the higher numbered select code has priority. With no interrupt service routine in progress, any interrupt will be granted.

INTERRUPT POLLS

Devices request an interrupt by pulling on one of two interrupt request lines ($\overline{IRL}$ and $\overline{IRH}$ - one for each priority level). The IOC determines the requesting select code by means of an interrupt poll, to be described in the next paragraph. If the IOC grants the interrupt it saves the existing select

THE INTERRUPT SYSTEM

INTERRUPT POLLS (CONT.)

code located in PA, puts the interrupting select code in PA, and does a JSM-Indirect through an interrupt table to get to the interrupt service routine.

An interrupt poll is a special I/O Bus Cycle to determine which interface(s) is (are) requesting an interrupt. An interrupt poll is restricted to one level of priority at a time, and is done only when the IOC is prepared to grant an interrupt for that level.

The interfaces distinguish an Interrupt Poll Bus Cycle from an ordinary I/O Bus Cycle through the $\overline{INT}$ line being low. Also, during this Bus Cycle $\overline{PAB3}$ specifies which priority level the poll is for. An interface that is requesting an interrupt on the level being polled responds by grounding the nth I/O Data line of the I/O Bus, where n equals the device's select code module eight. If more than one device is requesting an interrupt, the one with the higher select code will have priority.

The IOC has a three-deep first-in last-out hardware stack. The top of the stack is the Peripheral Address register (PA-11_8). The stack is deep enough to hold the select code in use prior to any interrupts, plus the select codes for two levels of interrupt. When an interrupt is granted, the IOC automatically pushes the select code of the interrupting device (as determined by the interrupt poll) onto the stack. Thus the previous select code-in-use is saved, and the new select code-in-use becomes the one of the interrupting device.

INTERRUPT TABLE

It is the responsibility of the firmware to maintain an interrupt table of 16 consecutive words, starting at some Read/Write Memory address whose four least-significant bits are zeros. The words in the interrupt table are set to the starting addresses of the various interrupt service routines in use for the 16 different select codes. When a peripheral is allowed to interrupt its select code is used to determine which interrupt service routine to JSM to. The interrupt service routine then handles the I/O operations needed by the interrupting device.

The firmware must also store the address of the first word of the interrupt table in the IV register (Interrupt Vector register, address 10_8, located in the IOC). Those contents will merge with the select code to produce the address of the appropriate table entry. In either version of the processor a two-level indirect jump is used to arrive at the interrupt service routine. This happens automatically because the BPC generates a JSM IV ,I as part of what it does during an interrupt. See Figures P-10 and P-10½. In 15-bit processors the indirect chain could be longer if desired. It cannot be shorter, however, due to a bug in the 15-bit IOC. Thus, the scheme depicted in Figure P-11 cannot be used. Even with 16-bit processors the scheme in Figure P-11 is not possible; in 16-bit processors the IOC forces the BPC to do two consecutive "first-level" indirect accesses, so that the effect is exactly that shown in Figure P-11, except that it doesn't matter then whether bit 15 of IV is set or not.

In 15-bit processors bit 15 of IV <u>must</u> be set. This does two things. First, it guarantees that the JSM 10_8,$\overline{I}$ involves at least two levels of indirect. Second, it avoids a bug in the IOC. If bit 15 were a zero, the machine would attempt to implement the situation shown in Figure P-11. But

THE INTERRUPT SYSTEM

INTERRUPT TABLE (CONT.)

a race condition between the BPC and IOC is involved; its effect is to make bit 15 of IV look like a one even when it was a zero. The bug is somewhat dependent upon clock frequency. Reliable operation can be ensured only by using a two-level JSM through IV and the interrupt table.

In 16-bit processors the bug was fixed by permanently deciding the race condition in the IOC's favor. Nothing was done to the BPC; it still only understands one level of indirect addressing. But the IOC keeps the INT line grounded long enough to force the BPC to treat the contents of IV itself as an *indirect address*. This causes the BPC to read the next address (the one in the interrupt table) and treat *its* contents as the destination address, just as in multi-level indirect addressing. Thus, in the 16-bit processor the JSM through the interrupt table is always a two-level process as shown in Figure P-10½, regardless of whether bit 15 of IV is set or not. Bit 15 of IV becomes simply an address bit, helping indicate where in memory the interrupt table is located.

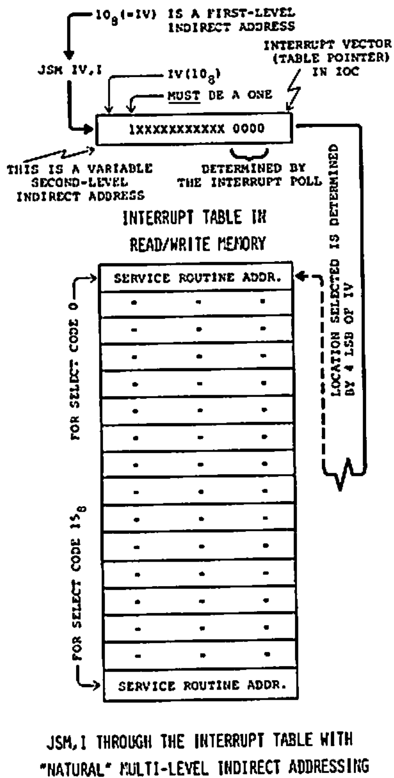


Figure P-10.
The Interrupt Table With 15-Bit Addressing.

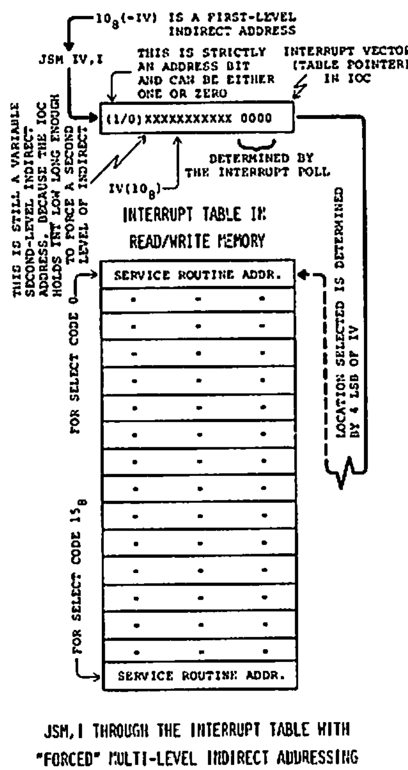


Figure P-10½.
The Interrupt Table With 16-Bit Addressing.

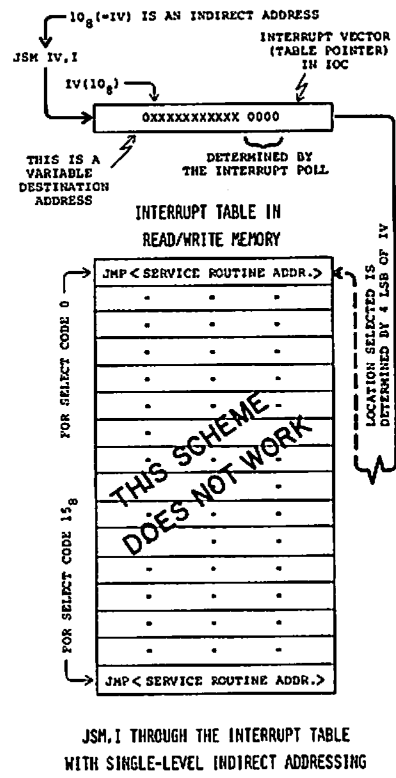


Figure P-11.
How Not To Use The Interrupt Table.

After the interrupt poll is complete the select code of the interrupting device is made to be the four least-significant bits of the IV register. Thus IV now points at the word in the Interrupt Table which corresponds to the appropriate interrupt service routine. All that is needed now is a JSM IV,I,

THE INTERRUPT SYSTEM

INTERRUPT TABLE (CONT.)

and the interrupt service routine will be under way. This is accomplished by the BPC as summarized below.

INTERRUPT PROCESS SUMMARY

The IOC inspects the interrupt requests $\overline{IRL}$ and $\overline{IRH}$ during the time sync is given. Based on the priority of the interrupt requests, and the priority of any interrupt in progress, the IOC decides whether or not to grant an interrupt. If it decides to allow an interrupt it immediately pulls $\overline{INT}$ to ground, and also begins an interrupt poll.

The grounding of $\overline{INT}$ serves three purposes: It allows the interfaces to identify the forthcoming I/O Bus Cycle as an interrupt poll; it causes any other chips in the system, except the BPC, to abort their instruction decode process (which by this time is in progress) and return their idle states; and it causes the BPC to abort its instruction decode and execute a JSM 10_8,I instead.

The IOC uses the results of the interrupt poll to form the interrupt vector, which is then used by the JSM 10_8,I. It also pushes the new select code onto the peripheral address stack, and puts itself into a configuration where all interrupt requests except those of higher priority will be ignored.

INTERRUPT SERVICE ROUTINES

The majority of the interrupt activity described so far is accomplished automatically by the hardware. All the firmware has been responsible for has been the IV register, the maintenance of the interrupt table, and (probably) the initiation of the particular peripheral operation involved (plotting a point, backspace, finding a file, etc.). Such operations (initated through a command given by simple programmed I/O) may involve many subsequent I/O Bus Cycles, done at odd time-intervals, and requested by the peripheral through an interrupt. It is the responsibility of the interrupt service routine to handle the I/O activity required by the peripheral without upsetting the routine that was interrupted.

It's difficult to say specific things about interrupt service routines in general; a lot depends upon the particulars of the host software system. In the next few pages we will examine some generalities relating to interrupt service routines, and sketch some examples. The result may leave some readers with an unsatisfied feeling; specific information is not available except as part of a description of a particular software system.

Our first observation is on the number of service routines. In general, there is not one service routine for each select code, or even for each peripheral. The usual case is collections of routines that perform related functions within the needs of a certain class of peripheral activity; each class of activity has it own collection.

THE INTERRUPT SYSTEM

INTERRUPT SERVICE ROUTINES (CONT.)

For instance, it is unlikely that there would be a single interrupt service routine for a disc. On the customer's level there are many commands in the disc operating system. On the firmware level there are a series of routines that perform 'fundamental units' of activity, where each 'fundamental unit' involves some amount of I/O. Most commands in the user's disc operating system are made up of a series of these 'fundamental units' of activity. 'Fundamental units' of activity for the disc are things like: moving the head to a given track, reading a given sector from a track into such and such a buffer, and writing from such and such a buffer into a given sector. It is these types of activity that are most likely to have corresponding interrupt service routines.

Let's sketch a hypothetical example. Assume a fairly involved disc user's command is to be performed, one requiring reading the directory on the disc to determine the location of certain file on the disc, and then loading that file into memory. The kind of thing that happens here is to move the head to the start of the directory, read through the information in the directory sector by sector until the information about the desired file is found, moving the head to the file's location, reading its header, reading its first sector, etc., etc.

Each service routine is smart enough to know which service routine follows it for the particular high level task at hand, and, if it has a choice based on the way events turn out (error conditions, etc.), it knows how to handle that, too. As each new step in the sequence requiring a different interrupt service routine is reached, the concluding routine changes the appropriate entry of the interrupt table to the starting address of the next service routine. In this way a versitile collection of interrupt service routines can serve many purposes.

As another example of this, consider a smart tape cassette, whose internal architecture was of variable length files composed of fixed length records. Such a cassette would resemble a disc from the user's point of view, and it is possible that some of the disc interrupt service routines would work for the cassette, also.

And lastly, consider the case of formatted output to line printers, punches, teletypes and CRT's. Some of these devices may differ slightly in their mainline firmware drivers, but there is an excellent chance that they could use the same general purpose interrupt service routine(s).

So much for the chicken, now for the egg. At the beginning of the operation the mainline firmware sets up any initial conditions that are required (e.g., selecting a buffer and setting a word count or a value of a pointer). The mainline firmware also selects the interrupt service routine by modifying an entry in the interrupt table. It also gives the first I/O Bus Cycle, which wakes up the peripheral and gets things going. After this first I/O Bus Cycle the mainline firmware can go on about its other business.

THE INTERRUPT SYSTEM

INTERRUPT SERVICE ROUTINE (CONT.)

Perhaps some questions have occured to you: "How does a peripheral know if it is supposed to interrupt, or operate in some other mode?" (A Low-end calculator might not use interrupt - or on a given calculator a peripheral may use interrupt sometimes but not others); "How is it that the routine that is in progress doesn't get bombed when an interrupt occurs?"; "And, come to think of it, how can the calculator proceed with other activity when it has essentially passed over unfinished business - might not things run amuck?"; and lastly, "How does the peripheral know when to stop interrupting, especially in the case of an output operation where an arbitrary amount of information is transmitted?"

HOW A PERIPHERAL KNOWS WHETHER TO USE INTERRUPT OR SIMPLE I/O

There are several possibilities here: it might never use interrupt; it might always use interrupt, it might use interrupt always with one mainframe but not with another due to different interface cards; it might have a smart interface card that knows what calculator it's in, and thus use interrupt or not; or, it might have a smarter yet interface that allows the calculator to tell the peripheral when to begin using the interrupt system, and when to stop.

The last possibility could work like this: The initial I/O Bus Cycle given by the mainline firmware could reference, say, R5. This would be understood by the interface as a command to interrupt as soon as the device is ready to handle the next ration of data. A scheme like this allows I/O statements referencing R4 free for simple, non-interrupt operation.

BOMBPROOFING THE MAINLINE FIRMWARE

The calculator could be almost anywhere in its internal coding when an interrupt is granted. Since the code is suspended with a JSM, the way is clear to get back to the right spot with a RET 0,P. But it won't do any good to come back it the items in memory related to the routine are not the same. The interrupt service routine must save and later restore any memory location that will be directly or indirectly disturbed by the activity of the service routine. This could include the extend and overflow registers of the BPC, decimal carry and shift-extend of the EMC, and possibly CB and DB in the 16-bit version of the IOC.

As long as the sevice routine does all its own laundry, it's easy to tell what to save; it's whatever gets used that's not private to that service routine. But if the service routine farms out some of its work to utility subroutines in the main system, what needs to be saved is not always obvious.

"SIMULTANEOUS" ACTIVITIES

The main system software must be designed with interrupt in mind to take full advantage of the interrupt system. This generally involves an entirely different approach to I/O than in less sophisticated machines where there is no interrupt capability. The following example illustrates the sort of

THE INTERRUPT SYSTEM

"SIMULTANEOUS" ACTIVITIES (CONT.)

approach used with interrupt systems.

Consider the following program segment:

```
50          WRITE (6,100)   A,B,C,
55          X = (A + B + C)/3
60          A = A + 1
            :
            :
            :
100         FORMAT I10, 2F20.5
```

The write statement of line 50 is to be done under interrupt. Basically, the idea is that once the firmware that executes the write statement has gotten things started, the calculator can begin to execute the next line in the program. In this example it is safe to immediately execute line 55, as it will not affect the on-going process for line 50. But line 60 is another matter. Whether or not it is safe to execute line 60 depends upon how the main system works.

Suppose the main system has lots of memory to burn, and that the WRITE routine, as part of its initialization, went and got the values of A, B, and C, and saved them in a buffer. Then nothing can hurt line 50; line 60 can be executed immediately.

On the other hand, consider a system with not so much memory, and consequently, little or no buffering. It could compromise by setting a bit in the symbol table entries of A, B, and C, marking them as busy. As each is outputted, it would be un-marked. Then line 60 would be executed if A were not busy, or, there would be a delay at line 60 while the main system waits for A to become non-busy.

WHEN TO CEASE INTERRUPT MODE OPERATION

In some cases the peripheral and the corresponding firmware may each know in advance how many items are involved, and each just goes to sleep when everything is done.

In the case of arbitrary length transfers, or transfers controlled by one party, however, somebody has to decide when it's all done, and notify the other party. For most output operations, and for input operations involving dumb peripherals, the smarts are in the firmware. What the peripheral will do is interrupt as soon as it is available following the exchange of some data, even if the previous exchange was the "last" one (which the peripheral didn't know). It will do this, unless the interrupt mode in the interface is shut off before it has the chance to interrupt again.

THE INTERRUPT SYSTEM

WHEN TO CEASE INTERRUPT MODE OPERATION (CONT.)

Now for hardware reasons the peripheral will, while requesting an interrupt, keep its Interrupt Request line active until it gets a (data) I/O Bus Cycle for that device.* The consequences of this are that once the interrupt is granted the interrupt service routine cannot decline to exchange more data and terminate itself by simply executing only a RET 0,P. To do so would leave the interface thinking it never got recognized (no data I/O Bus Cycle), while the IOC thinks the interrupt is over. So on the next instruction fetch the interrupt is granted again!! (Assuming the priority situation has not changed.)

So, unless the device is smart enough to know, by itself, not to interrupt after the last exchange, the firmware must shut the thing off. This easy enough to do, and could be done by taking advantage of the ability to set $\overline{IC1}$ and $\overline{IC2}$ during an I/O Bus Cycle (i.e., STA R5 or STA R6, perhaps with a special code in A). So the result is a different (and perhaps an extra) trailing I/O Bus Cycle to put the interrupt mode of the peripheral to sleep.

RETURNING FROM INTERRUPT SERVICE ROUTINES

The last things done by an interrupt service routine are to: (if necessary) shut off the interrupt mode of the interface; restore any saved values; and to execute a RET 0,P.

The RET 0 part acts to return to the routine that was interrupted, so that its execution will continue. The P acts to pop peripheral address stack and adjust the IOC's internal indicator of what priority level of interrupt is in progress. By popping the peripheral address stack, PA is set back to whatever it was prior to the most recent interrupt.

DISABLING THE INTERRUPT SYSTEM

The interrupt system can be "turned off" by a DIR instruction. After this instruction is given the IOC will refuse to grant any interrupts whatsoever, until the interrupt system is turned back on with the instruction EIR. While the IOC won't grant any interrupts, the RET 0,P works as usual so that interrupt service routines may be safely terminated, even while the interrupt system is turned off.

* It has to be this way because this is the only way a device requesting an interrupt can determine that it has been granted an interrupt. The mere doing of an interrupt poll for that level is not enough - a device on the same level but with a higher select code may be the winner. Nor can an interface tell if it is the winner by looking at the PA lines - the only signal usable as a strobe for that is given before they are set up.

FUNCTIONAL DESCRIPTION OF THE IOC

DIRECT MEMORY ACCESS

Direct Memory Access is a means to exchange entire blocks of data between memory and peripherals. A block is a series of consecutive memory locations. Once started, the process is mostly automatic; it is done under control of hardware in the IOC, and regulated by the interface.

The DMA process can transfer data in two ways: single words are transferred one at a time, on a cycle-steal basis; strings of words can be transferred consecutively in a burst mode. In either instance data is transferred one word at a time. To transfer a word, a peripheral signals the IOC, which then requests control of the IDA Bus with $\overline{BR}$. That results in an external halt in all other system activity on the Bus for the duration of the peripheral's request for DMA service. Herein lies the difference between burst mode and cycle-steal operation; in cycle-steal operation the peripheral ceases to request service after one word is transferred, and requests service again when ready, while in the burst mode the request is held to allow a series of high-speed consecutive transfers to occur.

During a DMA transfer of a block of data the IOC knows the next memory location involved, whether input or output, which select code, (and possibly) whether or not the transfer of the entire block is complete. This information is in registers in the IOC, which are set up by the firmware before the peripheral is told to begine DMA activity.

Actual transfers are initiated at the request of the interface. To request a DMA transfer a device grounds the DMA Request line ($\overline{DMAR}$). Since there is only one channel of DMA hardware, and one DMA Request line, only one peripheral at a time may use DMA. A situation where two or more devices compete for the DMA channel must be resolved by the firmware, and it is absolutely forbidden for two or more devices to ground $\overline{DMAR}$ at the same time. (A data request for DMA is not like an interrupt request; there is no priority scheme, and no means for the hardware to select, identify and notify an interface as the winner of a race for DMA service.) Furthermore, a device must not begin requesting DMA transfers on its own; it must wait until instructed to do so by the firmware.

The DMA process is altogether independent of the operation of standard I/O and of the interrupt system, and except for theft of the IDA Bus for memory cycles, does not interfere with them in any way.

ENABLING AND DISABLING THE DMA MODE

DMA transfers as described above are referred to as the DMA Mode. The DMA Mode can be disabled two ways: by a DDR (Disable Data Request), or by a PCM (Pulse Count Mode - described later). A DDR causes the IOC to simply ignore $\overline{DMAR}$; no more, no less. The instruction DMA (DMA Mode) causes the IOC to resume DMA Mode operation; DMA cancels DDR, and vice versa. DMA also cancels PCM, and vice versa. Also, DDR cancels PCM, and vice versa.

Also, the IOC turns on as if it has just been given a DDR. DDR (along with DIR) is useful during system initialization (or possible error recovery) routines, where it is unsafe to allow any system activity to proceed until

DIRECT MEMORY ACCESS

ENABLING AND DISABLING THE DMA MODE (CONT.)

the system is properly initialized (or restarted).

REGISTER SET-UP

There are several registers that must be set up prior to the onset of DMA activity. These are shown below:

Name	Address	Meaning
DMAPA	$(=13_8)$	DMA Peripheral Address
DMAMA	$(=14_8)$	DMA Memory Address (and direction for 15-bit addressing)
DMAC	$(=15_8)$	DMA Count
DMAD	------	DMA Direction (for 16-bit addressing)

The four least significant bits of DMAPA specify the select code which is to be the peripheral side of the DMA activity. During an I/O Bus Cycle given in response to a DMA data request, the content of the PAB lines will be determined by the four least significant bits of DMAPA, rather than by the PA register.

DMAC can, if desired, be set to n-1, where n is the number of words to be transferred. During each transfer the count in DMAC is decremented. During the last transfer the IOC automatically generates signals which the interface can use to recognize the last transfer. In the case of a transfer of unknown size, DMAC should be set to a very large count, to thwart the automatic termination mechanism. In such cases it is up to the interface to identify the last transfer.

DMAMA is set to the address of the first word in the block to be transferred. This is the lowest numbered address; after each transfer DMAMA is automatically incremented by the IOC. For 15-bit addressing, bit 15 of DMAMA specifies input or output (relative to the processor); a zero specifies input and a one specifies output. With 16-bit addressing a separate one-bit register (DMAD) exists to specify the direction of the transfer; DMAD is controlled by its own set and clear instructions, and is not addressable.

DMA INITIATION

Once the control registers are set up, a "start DMA" command is given to the interface through standard programmed I/O. The "start DMA" command is an output I/O Bus Cycle with a particular combination of $\overline{IC1}$, $\overline{IC2}$, (and perhaps) a particular bit pattern in the transmitted word. The patterns themselves are subject to agreement between the firmware designer and the interface designer. Sophisticated peripherals using DMA in both directions will have two start commands, one for input and one for output. It's also possible that other information could be encoded in the start command (block size, for instance).

DIRECT MEMORY ACCESS (CONT.)

DATA REQUEST AND TRANSFER

The interface exerts $\overline{DMAR}$ low whenever it is ready to exchange a word of data. When $\overline{DMAR}$ goes low the IOC requests control of the IDA Bus. When granted the Bus, the IOC initiates an I/O Bus Cycle with the PA lines controlled by DMA Peripheral Address, and does a memory cycle. (The order of these two operations depends upon the direction of the transfer.)

Next the IOC increments DMA Memory Address and decrements DMA Count.

DMA TERMINATION

Both the 15-bit and 16-bit addressing processors employ an automatic DMA termination indicator that involves $\overline{IC2}$. The 15-bit version of the IOC contains an additional mechanism involving a signal called $\overline{CTM}$. Automatic termination is usable only when the block size is known in advance and is based on the count in DMAC going negative.

Recall that at the start of the operation DMAC is set to n-1, where n is the size of the transfer in words. During the transfer of the nth word, the IOC will signal the interface by temporarily exerting $\overline{IC2}$ high during the I/O Bus Cycle for that exchange. The interface can detect this and cease DMA operations.

The other means of automatic termination would be detection by the interface of a Count Minus signal ($\overline{CTM}$). $\overline{CTM}$ is generated by the 15-bit version of the IOC; it means that the count in the least significant 15 bits of DMAC has gone negative. $\overline{CTM}$ is a steady-state signal, given as soon as, and as long as, the count in DMAC is negative. While $\overline{CTM}$ is generated by the IOC, it proved unsatisfactory and it is not utilized in the configuration employed in the present 15-bit hybrid micro-processor. That is, $\overline{CTM}$ never leaves the IOC.

For DMA transferes of unknown block size, the interface determines when the transfer is complete, and flags or interrupts the processor.

THE PULSE COUNT MODE

The Pulse Count Mode is a means of using the DMA hardware to acknowledge, but do nothing about, some number of DMA requests. The Pulse Count Mode is initiated by a PCM, and resembles the DMA Mode, but without the memory cycle. The activities of the registers DMAPA, DMAC, DMAMA, and DMAD remain as described for DMA Mode operation. The *only* difference is that no data is exchanged with memory; no memory cycle is given. (The IOC even requests the IDA Bus, but when granted it, releases it without doing the memory cycle.)

A dummy I/O Bus Cycle is given, and DMAC decremented. Also, the automatic termination mechanism still functions; in fact, that is the object of the entire operation. The Pulse Count Mode is intended for applications like

DIRECT MEMORY ACCESS

THE PULSE COUNT MODE (CONT.)

the following: Suppose it were desired to move a tape cassette a known number of files. The firmware puts the appropriate number into DMAC, gives PCM, and instructs the cassette to begine moving. The cassette would give a DMA Request each time it encounters a file header. In this way the DMA hardware and the automatic termination mechanism count the number of files for the cassette. PCM cancels DMA and DDR. Both DMA and DDR cancel PCM.

PLACE AND WITHDRAW

THE NOTATION OF A STACK

A stack is a series of consecutive memory locations. A stack is treated as a unit of memory having a single 'depository' into which or from which all information in the stack must pass in a first-in, last-out, order. The depository is the 'top of the stack'. A stack that can contain one hundred words of information is one hundred words 'deep'.

Consider a 100 word stack containing one entry. That entry would be 'on top of the stack' and the remaining 99 words 'below' the top of the stack would be 'empty'. Suppose a second entry is made. Then this latest entry is on top of the stack, the first entry is just below it, and 98 empty words below that.

Data is removed from a stack in a way that is the reverse of the way it is put in: the top of the stack is deleted and the entries below 'move up' one location, with the entry formerly one below the top of the stack now becoming the new top of the stack.

Physically, a stack can be implemented in hardware or in firmware. In a genuine hardware stack all the entries actually move from their present locations to the next one, and, they all do it at the same time as a single operation. Obviously, this requires a considerable amount of interconnection between the locations in the stack.

A stack that is implemented in firmware is simply a series of consecutive memory locations, accessed indirectly through a pointer. Instead of the entries in the stack changing their physical locations in the memory during additions and deletions, the value of the pointer is incremented or decremented.

STACK OPERATIONS

The IOC includes some firmware stack manipulation instructions. Two registers are provided as stack pointers: C and D. There are eight place and withdraw instructions for putting things into stacks and getting them out. Furthermore, the place and withdraw instructions can handle full 16-bit words, or pack 8-bit bytes in words of a stack. And last, there are provisions for automatic incrementing and decrementing of the stack pointer registers, C and D.

PLACE AND WITHDRAW

STACK OPERATIONS (CONT.)

The mnemonics for the place and withdraw instructions are easy to decipher. All place instructions begin with P, and all withdraw instructions begin with W. The next character is a W or B, for word or byte, respectively. The next character is either a C or D, depending upon which stack pointer is to be used. There are eight combinations, and each is a legitimate instruction.

A PWD A,I reads as follows: place the entire word of A into the stack pointed at by D, and increment the pointer before the operation. The instruction WWC B,D is read: Withdraw an entire word from the stack pointed at by C, put the word into B, and decrement the stack pointer D after the operation.

The place and withdraw instructions outwardly resemble the memory reference instructions of the BPC: a mnemonic followed by an operand that is understood as an address, followed by an optional 'behavior modifier'. The range of values that the operand may have is restricted, however. The value of the operand must be between 0 and 7, inclusive. Thus, the place and withdraw instructions can place from, or, withdraw into, the first eight registers. These are A, B, P, R, and R4 through R7. Therefore, the place and withdraw instructions can initiate I/O Bus Cycles; they can do I/O.

The place and withdraw instructions automatically change the value of the stack pointer each time the stack is accessed. In the source text an increment or decrement is specified by including a ,I or a ,D respectively, after the operand.

Regardless of which of increment or decrement is specified, a place instruction will do the increment or decrement of the pointer prior to the actual place operation. Contrariwise, the withdraw instructions do the increment or decrement after actual withdraw operation. The reason for this is that it always leaves the stack with the pointer pointing at the new 'top-of-the-stack', and allows intermixing of place and withdraw instructions without adjustment of the pointer.

PLACE AND WITHDRAW FOR BYTES

One of the differences between the 15-bit and 16-bit versions of the processor is the way they handle byte operations for the place and withdraw instructions. Because the stack in memory is composed of words, rather than bytes, some means are required to extend the addressing of the pointer registers to include designation of bytes within the addressed word.

In 15-bit processors this is done with an unused bit in the pointer registers themselves; they are 16-bit registers while only 15-bits are needed to address the memory. Furthermore, the place and withdraw instructions do not allow a place or withdraw through C or D indirect. These conditions leave the left-most bit (bit 15) free to designate which byte (of the word at the top of the stack) is the byte in question. A one in bit 15 designates the left-half of the word at the top of the stack. It is up to the firmware to see that bit 15 is properly set prior to beginning stack operations.

PLACE AND WITHDRAW

PLACE AND WITHDRAW FOR BYTES (CONT.)

After each place or withdraw bit 15 is automatically toggled to provide a left-right-left-right sequence. During an automatic increment to the pointer register (,I) the address in the lower 15 bits increments during the zero-to-one transition of bit 15. Similarly, during an automatic decrement of the pointer register (,D) the transition of bit 15 from a one to a zero is accompanied by a decrement of the lower 15 bits.

The incrementing and decrementing schemes just described are only for increments and decrements brought about by a ,I or ,D following the operand of a Place or Withdraw instruction. Increments or decrements to the pointer register with ISZ or DSZ do not automatically toggle bit 15.

In 16-bit processors left-right indication of bytes is accomplished with a signal called $\overline{BL}$; there is no unused address bit as in 15-bit addressing. $\overline{BL}$ (Byte Left Not) is in turn controlled by bit 0 of either the C or D registers, as shown in Figure P-12. Sixteen-bit addressing is maintained by providing an additional one-bit register for use with each stack pointer register. The non-addressable registers are called CB (C Block) and DB (D Block). They are designated "block" because, as the most-significant bit of the word pointer value, they divide the address space into two halves, or "blocks".

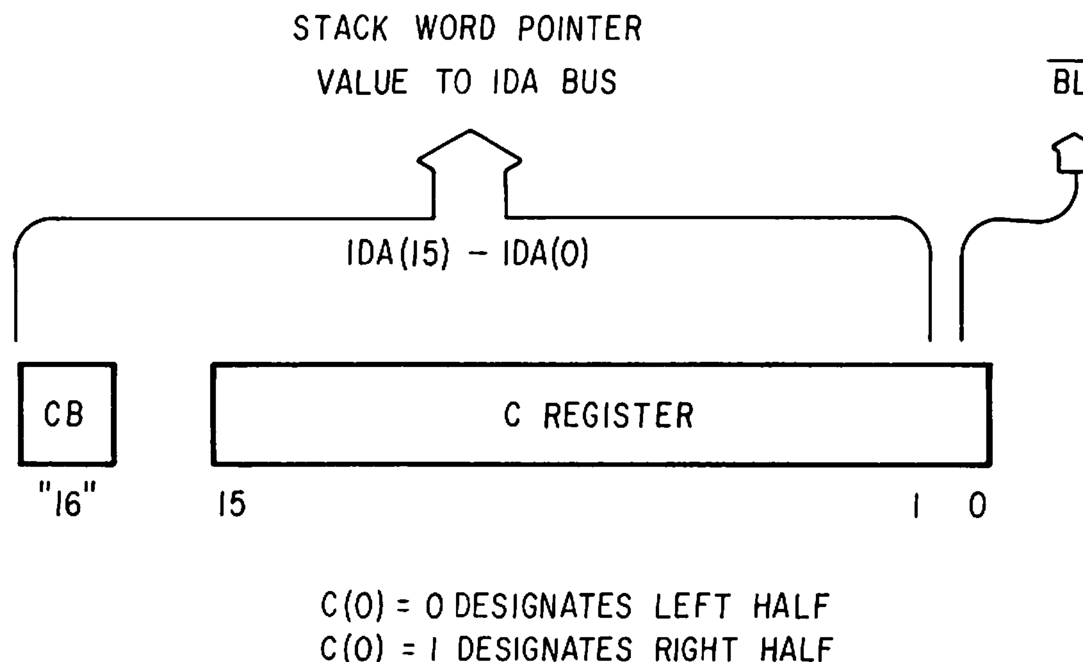


Figure P-12. Sixteen-Bit Stack Pointer Addressing.

PLACE AND WITHDRAW

PLACE AND WITHDRAW FOR BYTES (CONT.)

Figure P-12 shows how CB is used with C for place-byte and withdraw-byte operations that use the C register as the stack pointer. For such operations that use the D register instead, DB acts as the most-significant bit of the address, and bit 0 of D controls $\overline{BL}$.

During the automatic increment or decrement to the pointer register, CB and DB function as most-significant 17th bits for their respective registers. An advantage of having the bit that designated the byte be the least-significant bit is simplification of the process of arithmetic computation upon byte-addresses.

The CB and DB registers can be set to their initial values by machine-instructions for setting and clearing each register. For instance, DBU (D Block Upper) sets the DB register; CBL (C Block Lower) clears the CB register.

During the execution of a program the current values of CB and DB can be obtained by reading the contents of the DMAPA Register (13_8). While the four least-significant bits are the select code of a DMA-related peripheral, bit 15 reflects CB and bit 14 reflects DB. A one stands for upper, while a zero means lower. See Table P-1. Please note that CB and DB cannot be altered by writing into register 13_8; such alteration must be done by using the machine-instructions mentioned in the previous paragraph. If, for instance, an interrupt service routine involves the use of place or withdraw byte instructions, the service routine would need to save and later restore the initial values of whichever block-pointers were used (CB & DB), as well as set them up for use within the routine itself.

The place-byte instructions cannot be used to place bytes into the registers within the BPC, EMC and IOC. The reason for this is that these chips do not utilize the $\overline{BYTE}$ line of the IDA Bus during references to their internal registers.

The $\overline{BYTE}$ line is a signal supplied by the IOC for use by any interested memory entity. The $\overline{BYTE}$ line indicates that whatever is being transferred to or from memory is a byte (8 bits) and that bit 15 of the address (for 15-bit processors) or BL (for 16-bit processors) indicates right or left half. During a write memory cycle it is up to the memory to merge the byte in question with its companion byte in the addressed word.

In the case of a withdraw-byte the memory can supply the full 16-bit word (that is , ignore the $\overline{BYTE}$ line). The IOC will extract the proper byte from the full word and store it as the right-half of the referenced register; the left-half of the referenced register is cleared. In the case of a place-byte, however, the IOC copies the entire referenced register into an internal working register (W), and outputs its right-half as either the upper of lower byte (according to bit 15 of the address) in a full 16-bit word. The full word is transmitted to the memory, and the "other" byte is all zeros. Thus, in this case the memory must utilize the $\overline{BYTE}$ line.

PLACE AND WITHDRAW

PLACE AND WITHDRAW FOR BYTES (CONT.)

The consequence of the above is that any byte-oriented stacks to be managed using the place instructions must not include registers in any of the BPC, EMC, or IOC; that is, C and D must not assume any value between 0 and 37_8 inclusive for a place-byte instruction.

> **NOTE**
>
> An anomaly has been discovered in the operation of the IOC. If, while $\overline{\text{BYTE}}$ is low in conjunction with a memory cycle which is in progress, a Bus Request occurrs, then $\overline{\text{BYTE}}$ may pulse high for 10-40 nsec at the beginning of each Ø2, for the duration of that memory cycle. The severity of the glitch is related to the inherent speed of the chip, and to the exact timing relationship between Ø1 and Ø2. There doesn't appear to be any way to avoid the glitch, and therefore it maybe necessary for the designer to design around it.

INITIALIZATION OF TURN-ON

There is a signal called $\overline{\text{POP}}$ which is generated by the power supply. Its function is to prevent the chips from running except when power supply conditions are adequate. Chips can use $\overline{\text{POP}}$ to initialize certain internal conditions upon turn-on. The IOC does this. After turn-on the interrupt and DMA systems are left in the disabled state. The contents of the internal registers are random.

In the 15-bit version $\overline{\text{POP}}$ is held low by the power supply until all voltages have stabilized. Then $\overline{\text{POP}}$ is pulled high at the beginning of a Ø2.

In the 16-bit version $\overline{\text{POP}}$ synchronizer circuit was added to each chip. The intent is to free $\overline{\text{POP}}$ from synchronous phasing restrictions. The only requirement is that $\overline{\text{POP}}$ transition sharply to avoid threshold ambiguities in the various synchronizers. Unfortunatly however, some trouble has been experienced with this scheme. At least one designer has claimed flatly that the new scheme does not work, and that the old synchronous-with-Ø2 rule must still be observed.

GENERAL INFORMATION ABOUT THE EMC

The Extended Math Chip (EMC) provides 15 instructions. Eleven of these operate on BCD-coded three-word mantissa data. Two operate on blocks of data of from 1 to 16 words. One is a binary multiply and one clears the Decimal Carry (DC) register.

Unless specified otherwise, the contents of the registers A, B, SE and DC are not changed by the execution of any of the EMC's instructions.

The EMC communicates with other chips along the IDA Bus in ways similar to how the IOC communicates via the Bus.

NOTATION

A number of notational devices are employed in describing the operation of the EMC.

The symbols <....> denote a reference to the actual contents of the named location. For instance:

$$<A> + <HOOK> \rightarrow A$$

represents the instruction ADA HOOK.

A_{0-3} and B_{0-3} denote the four least significant bit-positions of the A and B registers, respectively. Similarly, A_{4-15} denotes the 12 most-significant bit-positions of the A register. And by the previous convention, $<A_{0-3}>$ represents the bit pattern contained in the four least-significant bit-positions of A.

AR1 is the label of a four-word arithmetic register located in R/W memory, locations $(1)77770_8$ through $(1)77773_8$. The assembler (ASMA) pre-defines the symbol AR1 as address 77770_8 (for 15-bit assemblies), or as address 177770_8 (for 16-bit assemblies).*

AR2 is the label of a four-word arithmetic accumulator register located within the EMC, and occupying register addresses 20_8 through 23_8. ASMA pre-defines the symbol AR2 as address 20_8.

SE is the label for the four-bit shift-extend register, located within the EMC. Although SE is addressable, and can by read from, and stored into, its primary use is as internal intermediate storage during those EMC instructions that read something from, or put something into, A_{0-3}. ASMA pre-defines SE as 24_8.

DC is the mnemonic for the one-bit decimal-carry register located within the EMC. DC is set by the carry output of the decimal adder. Sometimes, in the schematic illustrations of what the EMC instructions do, we show DC as

* ASMA is the DOS-RTE assembler for CPD Processor. ASMA is a 2100-series-computer program, written in H-P Assembly Language.

NOTATION (CONT.)

being part of the actual computation, as well as being a repository for overflow. In such cases the initial value of DC affects the result. However, DC will usually be zero at the beginning of such an instruction. The firmware sees to that by various means.

DC does not have a register address. Instead, it is the object of the BPC instructions SDS and SDC (Skip if Decimal Carry Set and Skip if Decimal Carry Clear), and the EMC instruction CDC (Clear Decimal Carry).

DATA FORMAT

The EMC can perform operations on twelve-digit, BCD-encoded, floating-point numbers. Such numbers occupy four words of memory, and the various parts of a number are put into specific portions of the four words, as shown in Figure P-13. The twelve mantissa digits are denoted by D_1 through D_{12}. D_1 is the most-significant digit, and D_{12} is the least-significant digit. It is assumed that there is a decimal point between D_1 and D_2.

ADDRESS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	E_S	TWO'S COMPLEMENT EXPONENT									EMPTY					M_S
M + 1	D_1				D_2				D_3				D_4			
M + 2	D_5				D_6				D_7				D_8			
M + 3	D_9				D_{10}				D_{11}				D_{12}			

Figure P-13. Floating-Point Data Format.

E_S and M_S each represent positive and negative (signs) by zero and one, respectively.

Those unfamiliar with two's complement arithmetic, and possibly the general procedures of firmware-implemented arithmetic, will find a modest explanation in the next section: A Beginner's Look at Calculator Arithmetic.

A BEGINNER'S LOOK AT CALCULATOR ARITHMETIC

This survey of arithmetic techniques is offered as an introduction for those not familiar with them. It doesn't cover the entire subject, nor is it always rigorous. Methods of implementing arithmetic differ widely, and the best we can do is tip our hats to some fundamentals, and to some general approaches. We will, however, be able to explain certain hardware features of the BPC and EMC that are related to arithmetic, as well as why certain instructions are fashioned as they are.

NUMERICAL REPRESENTATIONS

If someone were to ask you Jack Benny's age, you would immediately answer, "Why, thirty-nine, of course."* You probably wouldn't say:

a. one-oh-oh-one-one-one

b. oh-oh-one-one, one-oh-oh-one

c. ex-ex-ex-eye-ex

As humans, we have developed a "natural" method of representing numbers by using combinations of ten symbols, and we call it the decimal system. It works fine for calculations done mentally, with pencil and paper or other computing aids, and for the internal goings-on of the ferocious and many-toothed monster, the mechanical adding machine. Unfortunately, the decimal system is not directly implementable inside calculators or computers.

BINARY

You are no doubt familiar with binary and octal, and know that there are conversion processes for converting numbers expressed in a given base to any other base. The natural appeal of binary for computing mechanisms is irresistible, because its two digits one and zero so nicely match existing technology, and because it does not require complex circuitry to implement.

Table P-3.

COMPARISON OF DECIMAL, BINARY, AND OCTAL

DECIMAL	BINARY	OCTAL	DECIMAL	BINARY	OCTAL
0	0	0	6	110	6
1	1	1	7	111	7
2	10	2	8	1000	10
3	11	3	9	1001	11
4	100	4	10	1010	12
5	101	5	11	1011	13

* The Great Jack Benny is now the Late Jack Benny. Your author had the bad judgment to hang this example on someone who was <u>really</u> 1010000 years old.

NUMERICAL REPRESENTATIONS

BINARY (CONT.)

Binary is an arabic number system* (as is decimal), producing carries during addition, and allowing a binary point for writing fractional parts of a number. In fact, pencil and paper arithmetic can be done on binary numbers using exactly the same general procedures as for decimal numbers -- simply use binary addition and multiplication tables.

Still, there is always a fly in the ointment. It's not likely that the customer will be willing to key in his data using binary. This necessitates conversion; a distasteful process to many. What's more, many fractions that can be represented exactly in decimal cannot be represented exactly in binary (e.g., $.1_{10} = .0001100011\ \ldots_2$). [Lest you assume that there is something wrong with binary, the same thing happens in decimal: 1/7 = .1428571428571 ....]

For these and other reasons, representing numbers directly in binary in HP calculators is usually limited to cases where it is easy to do so, few arithmetic computations other than addition and subtraction are required, and to where the numbers involved are apt to be integers.

BINARY-CODED DECIMAL

The customer's numbers do get encoded, but in our case, into binary-coded decimal (BCD). Not only that, but the elements of the resulting code are arranged in a floating-point format. BCD is the familiar scheme of using four-bit binary codes in place of the decimal digits. Thus a 12-digit integer can be represented by 48 bits. In addition, the use of floating-point conventions adds sign information, and greatly enhances the maximum and minimum sizes of the numbers that can be encoded.

THE BCD DIGITS			
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

* An arabic number system is one in which a number is expressed as the sum of multiples of successive integer powers of a number n (called the radix), using n digits; 0, ... , n-1:

$$x = \ldots d_2 n^2 + d_1 n^1 + d_0 n^0 + .d_{-1}/n + d_{-2}/n^2 \ldots$$

(radix point indicated at the "." before d_{-1})

There are other schemes for representing numbers, such as the (abominable) roman numeral system. Multiplication is reportedly very difficult in that system.

NUMERICAL REPRESENTATIONS

BINARY-CODED DECIMAL (CONT.)

ADDRESS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0
M + 1	0011				0101				1000				0111			
M + 2	0010				0001				1001				0000			
M + 3	0000				0000				0000				0000			

Figure P-14. The Internal Floating Point Representation of .003587219 (= 3.587219×10^{-3}).

While BCD does allow exact representations of the original things the customer keys in (unless he is in the habit of keying in fractions like 1/7), BCD gives rise to certain drawbacks. First, BCD is wasteful of bits. Each four-bit combination can encode 16 symbols, while only 10 of these are ever used. The net result is that it takes more bits to encode numbers in BCD than it does to represent them directly in binary. (You could even have floating-point binary numbers if you wanted to.) The second thing is that BCD is indeed just a code, and not in itself an arabic numbering system. In general, you cannot add two BCD integers, bit-by-bit, and expect the result to be the correct (or even another) BCD number.

It takes a special gear works to handle BCD numbers. Done in firmware alone, such a gear works would be slow and cumbersome. The EMC supplies some useful operations on portions of BCD floating-point numbers. This trims the gear works in size, and speeds it up by quite a bit.

BINARY ARITHMETIC

Both the BPC and EMC have binary arithmetic capabilities. The BPC has binary add and complement instructions, while the EMC has a binary multiply instruction.

BINARY COMPLEMENTS

The BPC provides instructions for doing two kinds of complements: two's complements with TCA and TCB, and one's complements with CMA and CMB.

The one's complement of a binary number is its bit-by-bit complement. Another way to express this is to say that the number is subtracted from all one's, or if the number has n digits, from 2^n-1.

BINARY ARITHMETIC

BINARY COMPLEMENTS (CONT.)

```
 111                                    1111111
-101  <------------------------.       -0101011  <------------------------.
 010  <-- IS 1'S COMPLEMENT OF-'        1010100  <-- IS 1'S COMPLEMENT OF-'
```

With the CPD processor, one's complements are not used in arithmetic, but do find use in logical operations.

The two's complement of an n-bit binary number can be found in two ways: (1), by adding one to the one's complement; or (2), by subtracting the number from 2^n.

```
   111111          1000000
 - 101011        -  101011
   010100           010101
 +      1
   010101
```

The CPD processor does use two's complements in binary arithmetic. The notion of a two's complement does two things: first, it provides a compact and useful method of representing negative numbers*; second, it removes the need for a subtraction gear works in the hardware.

The use of the (signed) two's complement form to represent negative numbers has additional advantages: it eliminates the frequent need to recomplement an answer after a summation between numbers with different signs; and it automatically generates the proper sign in the answer (assuming no overflow).

These are significant advantages, not to be taken lightly. If you will, take a moment and consider algegraic BCD summations:

The need to re-complement occurs often in BCD arithmetic as performed by the CPD processor. In those cases numbers are always represented in uncomplemented form, regardless of sign. Numbers are complemented only to allow summations between numbers whose signs are different. After such a summation it is necessary to complement the answer if no "overflow" occurred. If overflow did occur, then everything is alright, and the "overflow" is ignored. Also, special attention must be given to the sign of the result.

* There are other compact methods of representing negative numbers. One such is sign-magnitude. There a single bit, say the most significant one, represents the sign, while the least significant bits always represent to absolute value of the number (its magnitude). By and large it is not as handy as two's complement representation. It requires either a hardware subtraction gear works, or extended handling in firmware, as described for BCD in a few paragraphs.

BINARY ARITHMETIC

BINARY COMPLEMENTS (CONT.)

As you read the next section, describing two's complement arithmetic, don't associated the "overflow" of the previous paragraph with binary overflow as discussed for two's complement arithmetic. They are not the same thing. The "overflow" for BCD arithmetic is simply a carry-out from the left-most digit, which results in DC (Decimal Carry) being set. The corresponding thing in our binary arithmetic is the setting of the E (Extend) register whenever there is a carry-out from bit 15. Binary overflow (the setting of OV) is a much more sophisticated condition.

TWO'S COMPLEMENT SUMMATION

Signed two's complement arithmetic in 16 bits limits the value of a single precision (one word) binary number to the range $+2^{15}-1$ (15 ones) through -2^{15} (a one followed by 15 zeros).

(+1) =	0000000000000001	(-1) =	1111111111111111
(+2) =	0000000000000010	(-2) =	1111111111111110
(+3) =	0000000000000011	(-3) =	1111111111111101
(+32767) =	0111111111111111	(-32767) =	1000000000000001
(±0) =	0000000000000000	(-32768) =	1000000000000000

In the above examples, the left-most bit serves as a sign bit, as well as a part of a complemented (and thus negative) number. Any number whose bit 15 is zero is a positive number and any number whose bit 15 is one is a negative number. The range limitation mentioned in the preceeding paragraph arise from there being only 15 bits (0-14) available to represent magnitudes of individual numbers.

Even though signed two's complement representation is often thought of as 15 bits of true-form or complement-form number, preceeded by a sign bit, the actual hardware mechanism that does the signed summations knows very little about signs or the two's complement format; it does a straight 16-bit binary add, with a carry out from bit 15 into the Extend (E) register. The only special property is the detection of overflow (results out of range); but even this only monitors events during summation, without changing them.

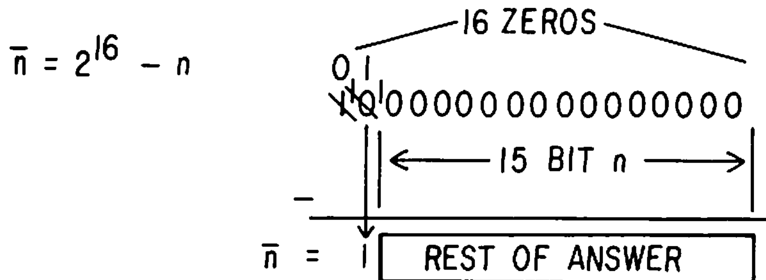
TWO'S COMPLEMENT SUBTRACTION

The rationale behind complement arithmetic is that the difference between two numbers can be found by the addition of one number to the complement of the other.

BINARY ARITHMETIC

TWO'S COMPLEMENT SUBTRACTION (CONT.)

The 16-bit two's complement of a 15-bit binary integer is:



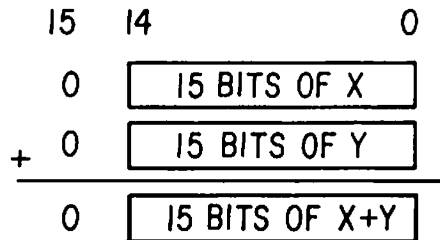
In a sense, $\bar{n}$ is the additive inverse of n:

$$(n + \bar{n}) \bmod 2^{16} = (n + (2^{16} - n)) \bmod 2^{16} = 2^{16} \bmod 2^{16} = 0$$

* * * * * * * *

The fact that two's complement arithmetic automatically produces the correct sign for the result is an important advantage, although it isn't at all obvious why it should be that way. The following demonstrations shows that correct answers are obtained.

Case I: X + Y (X > 0, Y > 0)



Both X and Y are positive. We assume that X and Y are such that their sum can be represented in 15 bits. Thus there is no possible carry out of bit 14, and the two bit 15's can only add up to zero, making the result positive.

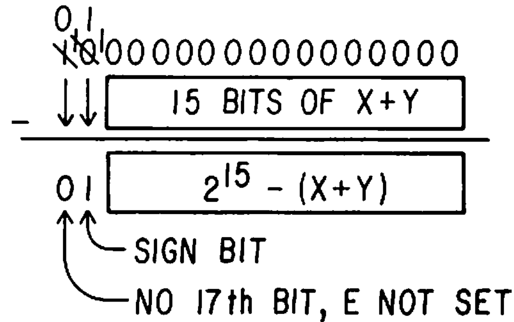
Case II: X + Y (X < 0, Y < 0)

For this case we note that -X-Y = -(X+Y)<0 which we complement and represent as 2^{16} - (X+Y). Once more we assume that X+Y does not exceed 15 bits.

BINARY ARITHMETIC

TWO'S COMPLEMENT SUBTRACTION (CONT.)

Case **II**: (cont.)



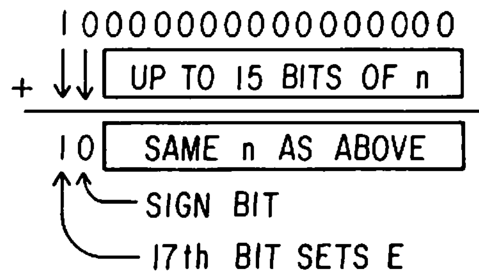
Because of the borrowing, the sign bit is a 1, and the answer is negative as we expect. We note also that a 1 preceeding 2^{15} - (X+Y) is the same as 2^{16} - (X+Y), which is the required answer.

Case **III**: X - Y (XY < 0)

$$X-Y = X + \overline{Y} = 2^{16} + X - Y$$

We can think of the terms + X-Y as some $n = |X-Y|$ which we add or subtract to 2^{16}, depending upon whether X > Y, or Y > X, respectively. (If X = Y, we can do either, since n = 0).

For X > Y:



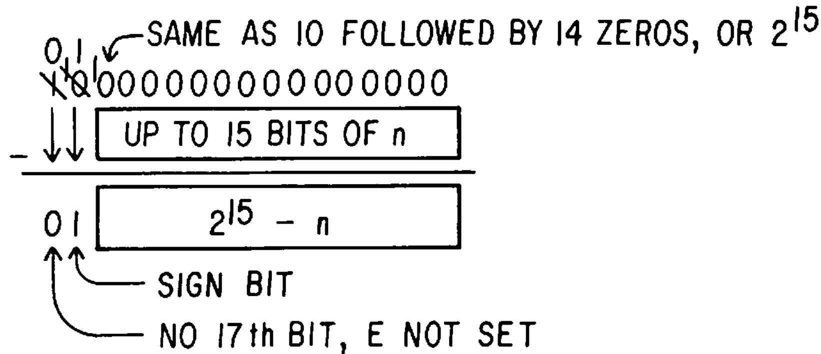
Here X > Y and n > 0, so n is added. Since each of the 15 bits of n is added to a zero bit, there can be no carries and the 16th bit (the sign) must be zero, also. This certainly agrees with X-Y > 0 when X > Y.

BINARY ARITHMETIC

TWO'S COMPLEMENT SUBTRACTION (CONT.)

Case III: (cont.)

But if X < Y:



Because of the borrowing, the sign bit is a 1. Thus the answer is negative, and this agrees with $X - Y < 0$ when $X < Y$. Finally, we should note that a 1 preceeding $2^{15} - n$ is the same as $2^{16} - n$, which is indeed the answer we set out to get.

By now you might be prepared to make the following objection: "The demonstration would be satifying, except that the hardware does not magically produce n, and then proceed to add it to, or subtract it from, 2^{16}; and, if it could do that, we probably wouldn't need two's complement arithmetic!"

True. The demonstration rests on the behavior of "equivalent" entities during "equivalent" operations. It is valid in that it does show that we don't ever get the wrong answer (assuming no binary overflow). But it doesn't give us any idea as to why it "really" works when the hardware adds up the bits.

We shall indulge in some quick examples that show how it *really* works.

First, consider the table of 5-bit two's complement numbers, on the next page.

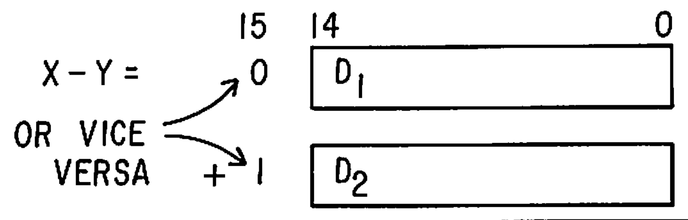
Consider 7-8. When the binary for 7 is added to the complement of 8, the result is the "biggest thing" that can fit into 5 bits, but there is no carry-out from the left-most bit. Looking at the table you can see that there is no carry-out for 7-n where $16 > n > 7$. Likewise, if $1 \leq n \leq 7$, n's complement is always big enough to generate a carry-out of the left-most bit.

BINARY ARITHMETIC

TWO'S COMPLEMENT SUBTRACTION (CONT.)

0 = 00000	8 = 01000	-1 = 11111	-9 = 10111
1 = 00001	9 = 01001	-2 = 11110	-10 = 10110
2 = 00010	10 = 01010	-3 = 11101	-11 = 10101
3 = 00011	11 = 01011	-4 = 11100	-12 = 10100
4 = 00100	12 = 01100	-5 = 11011	-13 = 10011
5 = 00101	13 = 01101	-6 = 11010	-14 = 10010
6 = 00110	14 = 01110	-7 = 11001	-15 = 10001
7 = 00111	15 = 01111	-8 = 11000	-16 = 10000

It is the carry-out of the left-most bit that is the vital clue. Consider 16-bit X and Y:



The sign bit (bit 15) will be a 1 (-) unless a carry is produced by the addition of the two bit 14's (d_1 and d_2). In fact, there will be a carry from bit 15 if and only if there is a carry from bit 14.

Suppose X > Y. Why must there be a carry? We are adding and get:

$$\underbrace{X} + \underbrace{2^{16} - Y} \geq 2^{16}$$

THESE ARE THE TWO BIT PATTERNS.

Think, if you wish, of the adder doing X increments to the bit pattern for 2^{16} - Y. Since X > Y, the effect of the -Y is entirely removed, causing a carry-out from bit 15. So we get carries out of both bits 14 and 15. This causes the sign to be positive, and sets 1 into the E register.*

Suppose Y > X. Then Y would absorb all of X before the sum reaches 2^{16}.

* E is generally ignored during binary arithmetic unless a multi-precision operation is in progress. See the Section after next.

BINARY ARITHMETIC

TWO'S COMPLEMENT SUBTRACTION (CONT.)

Thus there is no carry out of bit 14, and therefore none out of bit 15. The sign is negative and E does not get set.

TWO'S COMPLEMENT OVERFLOW

The conventions of signed two's complement arithmetic provide a useful method of detecting the generation of a result which is too large in magnitude to be represented in 16-bit signed two's complement form. We call this the overflow condition, and it occurs whenever there is a carry-out from bit 14, or, a carry-out from bit 15, but not if both carry-outs occur. The occurrence of the overflow condition sets the OV register to a one.

That the exclusive or of the carry-outs from bits 14 and 15 corresponds to the overflow condition is not at all obvious. There are three cases:

Case I: X + Y

Both numbers are positive. There can be no carry from bit 15. There is an overflow if and only if there is a carry from bit 14 (X and Y too big for a 15 bit sum).

	15	14 … 0
	0	UP TO 15 BITS OF X
+	0	UP TO 15 BITS OF Y

Case II: (-X) + (-Y)

	15	14 … 0
	1	15 BITS OF -X
+	1	15 BITS OF -Y

Both numbers are negative. There is always a carry from bit 15. Overflow results if and only if there is no carry from bit 14. Frankly, this is a tough one to properly explain by simply looking at the bits. So consider:

$$-X + (-Y) = 2^{16} - X + 2^{16} - Y = 2^{17} - (X + Y)$$

The maximum allowable size for X + Y without causing overflow is 2^{15}. This is shown by the three subtractions on the next page.

BINARY ARITHMETIC

TWO'S COMPLEMENT OVERFLOW (CONT.)

```
     0 1  14                   0
     X 0 0 0000000000000000 = 2^17
   - ↓ ↓ 1000000000000000   = 2^15 = X+Y
   ------------------------
     0 1 1
     ~~~ ↑
IGNORE THESE ↑  └─ SIGN BIT
```

```
     0 1 1
     X 0 0 0000000000000000 = 2^17
   - ↓ ↓ 0 1 --------------  = X+Y < 2^15
   ------------------------
     0 1 1
     ~~~ ↑
IGNORE THESE ↑  └─ SIGN BIT
```

The two subtractions above show that if (X + Y) is in range, a carry out result from bit 14 during the actual computation of -X-Y.* For the only way the sign bit in the answer could wind up a one is with a carry into bit 15. Likewise, it implies a carry out from bit 15, since both original bit 15's were ones to begin with. Both carries occurred, so there was no overflow.

Now suppose $X + Y > 2^{15}$. Here we get overflow.

```
     0 1 1
     X 0 0 0000000000000000 = 2^17
   -     1 ---------------- = X+Y > 2^15
   ------------------------
         0        ↖── AT LEAST ONE 1 SOMEPLACE
         ↑
         └── SIGN BIT
```

Because $X + Y > 2^{15}$, extra borrowing on the 2^{17} is necessary. This guarantees a zero in the sign bit of the result of the actual computation for -X-Y. Since the resulting sign bit is a zero, there could not have been a carry out of bit 14. Thus we are left with a solitary carry out of bit 15, (both original bit 15's were 1's, remember), and overflows results.

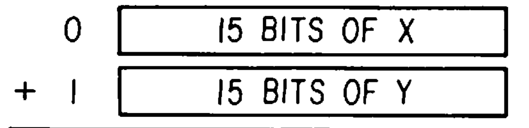
* We need to establish the link between the (positive) X + Y of our demonstration, and the (negative) -X-Y of the stated problem. This is easy, for if the limit on X + Y is 2^{15}, then: $X + Y = 2^{15} \rightarrow -(X + Y) = -2^{15} \rightarrow -X-Y = -2^{15}$. This comes as no surprise, as -2^{15} is the algebraically smallest number representable with 16-bit signed two's complement notation.

BINARY ARITHMETIC

TWO'S COMPLEMENT OVERFLOW (CONT.)

Case III: X-Y

The numbers have opposite signs. There can be a carry from bit 15 if and only if there is a carry from bit 14. That is, either both carries are present, or neither is present. The exclusive or condition can never be met.



MULTI-PRECISION BINARY ARITHMETIC

The main reason that the E register exists is to allow for the possibility of summations between binary numbers that are each two or more words in length. See Figure P-15.

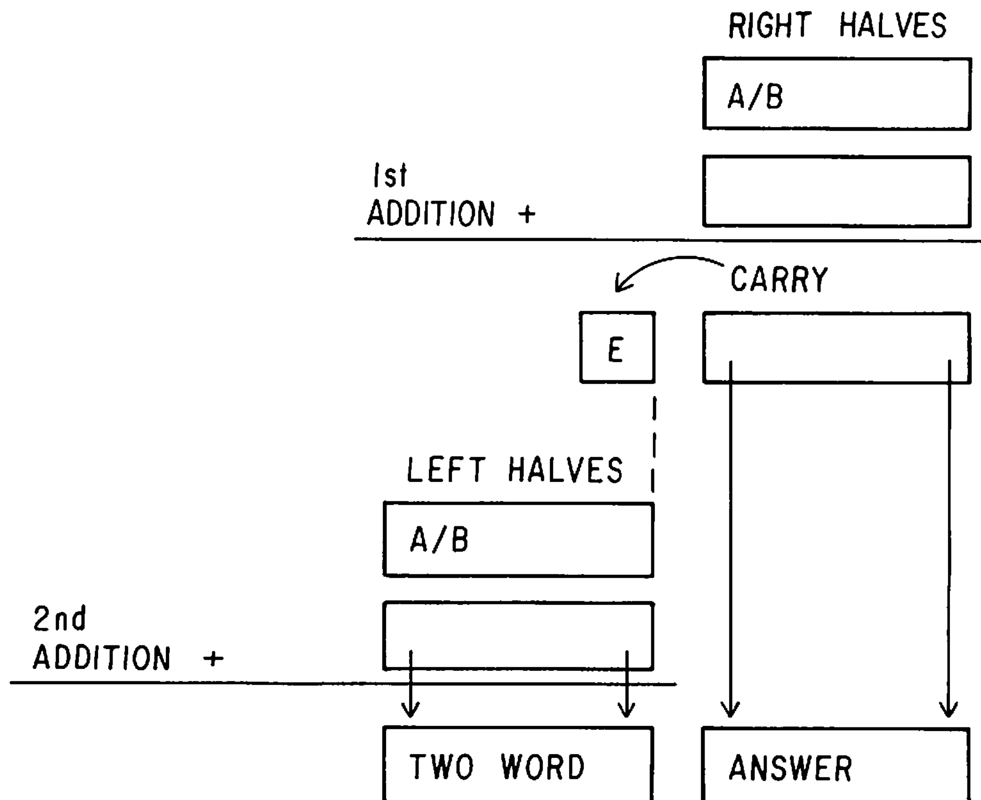


Figure P-15. Multi-Word Binary Addition Using the Extend Register.

MULTI-PRECISION BINARY ARITHMETIC (CONT.)

The scheme shown in Figure P-15 must be implemented in firmware; the ADA and ADB instructions do not automatically add in E. That must be done after testing with SES or SEC.

In multi-precision arithmetic, OV is ignored during all but the last addition, while E is checked after all but the last addition.

Complement arithmetic works perfectly well with multi-precision schemes. (Remember, ADA and ADB are full 16-bit adds.) Extra work is required to complement multi-word numbers, however, and cannot be done with just repeated applications of TCA or TCB. See Figure P-16.

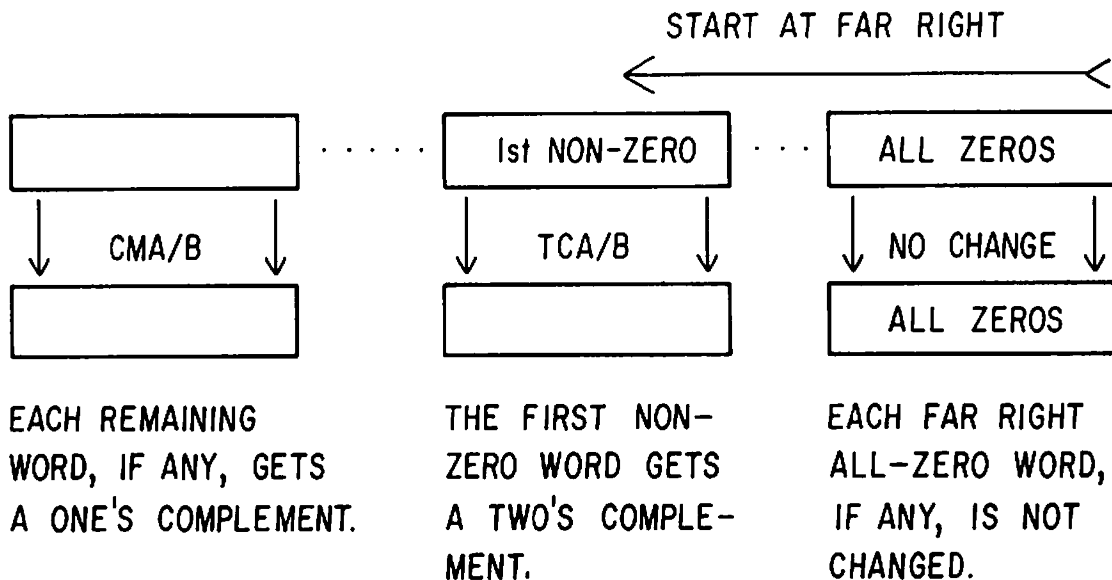


Figure P-16. Two's Complements of Multi-Word Binary Numbers.

Of course, it could also be done by simply doing a one's complement on each word, and then adding one to the result (using the multi-precision add).

ARITHMETIC SHIFTS

It sometimes happens that it is necessary to pack two's complement numbers of limited magnitude into fields within a word. An example is the exponent in the floating-point BCD format.

Assume that a copy of the exponent word is in A. Then an arithmetic right shift of six (AAR 6) will make the exponent in a proper 16-bit two's complement number.

ARITHMETIC SHIFTS (CONT.)

ADDRESS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	E_S	TWO'S COMPLEMENT EXPONENT									EMPTY					M_S
M + 1	D_1				D_2				D_3				D_4			
M + 2	D_5				D_6				D_7				D_8			
M + 3	D_9				D_{10}				D_{11}				D_{12}			

Figure P-17. Floating-Point Data Format.

Suppose the field labeled "empty" contained a 5-bit two's complement number. It could be made ready for use by an SAL 10 followed by an AAR 11.

The basis for this is that AAR and ABR propagate the sign while they shift the number. Consider the numbers ±3 in 5 bits, 10 bits, and 16 bits.

−3		+3
11101	←→	00011
1111111101	←→	0000000011
1111111111111101	←→	0000000000000011

Starting with A containing:

S1101-----------

An AAR 11 would produce:

PROPAGATED SIGN

SSSSSSSSSSSS1101

Arithmetic right shifts are provided for both the A and B registers.

BINARY MULTIPLY

The EMC provides a hardware implemented binary multiply for signed two's complement integers, using Booth's algorithm. See the description of the MPY instruction in the EMC MACHINE INSTRUCTIONS section for a complete definition.

Some explanatory material concerning the principles of Booth's algorithm is located in the Appendix.*

BCD ARITHMETIC

AR2 frequently functions as an accumulator for EMC operations on BCD numbers, much like the A and B registers are accumulators for the instructions ADA and ADB.

For the sake of completeness we will review some of the characteristics of the four-word packing formats for BCD numbers (see Figure P-17). The exponent and mantissa signs (E_S and M_S, respectively) are encoded as 0/1 for positive and negative, respectively. All of the digits D_1 through D_{12} are encoded in BCD, while the exponent is a 10-bit signed two's complement number. A decimal point is assumed to be between D_1 and D_2. D_1 is the most significant digit, and D_{12} is the least significant digit.

Except for intermediate results within the individual arithmetic operations, D_1 will never be zero unless the entire number is zero. Sometimes, after each individual arithmetic operation the answer needs to be *normalized*; that is, the digits of the answer shifted towards D_1 until D_1 is no longer zero. The exponent then needs to be adjusted to reflect the change.

The "empty" field of bits 1-5 in the exponent word is for possible future use in systems that allow different types of variable besides the full-precision real number which the present floating-point format accommodates. In such systems the "empty" field could contain a "type" indentifier, or some other information.

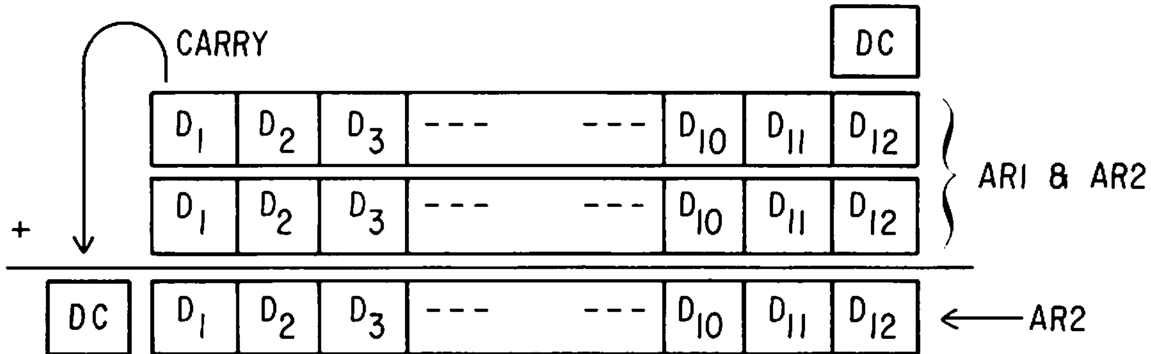
An important thing to keep in mind when examining BCD arithmetic, as implemented with the CPD processor, is that mantissas are represented in a sign-magnitude format. Ten's complements are used in the computational processes, but only as an intermediate step. Furthermore, it is done in such a way that the automatic generation of the correct sign of a sum does not occur. There is also the frequent need to re-complement an answer. All in all, BCD arithmetic is not as simple as two's complement binary arithmetic.

* For another explanation of Booth's Algorithm, refer to this book:
Digital Computer Design Fundamentals
Chu, Yaohan
McGraw-Hill (1962)
TK7888.3.C5

BCD ARITHMETIC (CONT.)

DECIMAL CARRY

The one bit Decimal Carry register (DC), located on the EMC, serves a function similar to that of OV for binary addition, although it is set by a rule similar to that for E.



DC is set to a one or zero, depending upon the occurrence or absense of a carry from the addition of the two D_1's, respectively. In this sense DC resembles E. But since the mantissas are represented in sign-magnitude form (with the sign in the exponent word rather than part of what gets added), DC also represents overflow for 12-digit mantissa additions.

Notice also that DC is part of the addition, in the D_{12} position. Frankly, this feature is seldom taken advantage of, if ever. It has potential use with multiple precision floating point arithmetic, and perhaps it will come in handy in some unknown future application.

There are three instructions that have to do only with DC. These are SDS (Skip if Decimal Carry Set) and SDC (Skip if Decimal Carry Clear) in the BPC instruction set, and CDC (Clear Decimal Carry) in the EMC instruction set.

TEN'S COMPLEMENT FOR BCD

The addition of the ten's complement of a number is used in lieu of a subtraction mechanism. If the signs of two numbers to be summed are different, one of the numbers is complemented (it doesn't really matter which one), before the addition.

The ten's complement of a 12-digit decimal integer X is:

$$\overline{X} \equiv 10^{12} - X$$

The ten's complement of a floating point number has the same exponent as the original number. The mantissa m of a floating point number fits the requirement:

$0 \leq m < 10$ (assuming the decimal point after D_1)

BCD ARITHMETIC

TEN'S COMPLEMENT FOR BCD (CONT.)

Therefore the complement of the mantissa alone is:

$$\overline{m} \equiv 10-m$$

Accordingly, all that is necessary to complement a floating point number is to complement the mantissa. It is immaterial whether the mantissa is treated as a 12-digit integer, or as a number between zero and ten; the same sequence of digits results.

Incidentally, here is a handy rule for finding the ten's complement of a decimal number: Ignore any right-most zero's--they stay the same. Subtract the right-most non-zero digit from ten, and those to the left of that, from nine.

As with two's complement, ten's complements are additive inverses, modulo 10^{12}:

$$X + \overline{X} = (X + (10^{12} - X)) \bmod 10^{12} = 10^{12} \bmod 10^{12} = 0$$

The EMC provides two instructions for doing ten's complements: CMX for AR1 and CMY for AR2. The only difference between these two instructions is that each operates upon a different "AR" register. What they do is replace each BCD digit, in the mantissa of the referenced register, with its appropriate digit of the complement.

Case I:

```
          0  9    9    9
10^12  =  1 10   10   10   10  00000000
          -  x    x    x    x  00000000
        -------------------------------
          0 9-x  9-x  9-x  10-x 00000000
         /  ↑                          ↑
       DC   D1                         D12
```

Case II:

```
          0 999999  9    9    9
10^12  =  1 101010101010 10   10   10   10  00
          - 000000  x    x    x    x  00
        ------------------------------------
          0 999999 9-x  9-x  9-x  10-x 00
         /  ↑                          ↑
       DC   D1                         D12
```

BCD ARITHMETIC

TEN'S COMPLEMENT FOR BCD (CONT.)

CMX and CMY leave the exponent word completely alone. This means that the sign of the mantissa, and the entire exponent are left unchanged in a ten's complement by CMX and CMY.

If you think about the above examples you'll see that we don't complement the actual floating point number in a Case II situation. For instance 3.561×10^{-4} complements to 6.439×10^{-4} if the mantissa is normalized. But sould the mantissa not be normalized, (and it frequently isn't when numbers are initially complemented - due to decimal point 'non-alignment'), the answer can be different. For instance, $.003561 \times 10^{-1}$ complements to 9.96439×10^{-1} when the mantissa is actually 003561. Now $.003561 \times 10^{-1} = 3.561 \times 10^{-4}$, but 9.96439×10^{-1} misses 6.439×10^{-4} by quite a ways.

Its puzzling at first glance, but it works. A good approach to BCD arithmetic is to treat the mantissa as an integer greater than or equal to zero, but less than 10^{12}. After all, if two numbers have equal exponents, it is strictly the sequence of digits in the two mantissas that determine the sequence of digits of the answer for any of the arithmetic operations. The exponent of the answer is determined by separate calculations involving only the exponents.*

It's making the exponents the same that causes the frequent "de-normalizing" of previously normalized floating-point numbers:

```
63,278 = 6.3278 E4 =  6.3278 E4
   531 = 5.31   E2 =+ .0531  E4
                     ----------
                      6.3809 E4 = 63,809
```

If you are willing to consider the mantissas by themselves, then its best to think of them as integers, as previously suggested, and pretend the decimal point is after D_{12}. Normalized mantissas are then represented by big integers: a one through nine followed by eleven other digits. A non-normal mantissa is simply a smaller integer by the extent it has zeros on the left. In two's complement representation the left-most zeros complement into ones; here they complement into nines.

There is a case III that we should mention:

Case III:

```
10^12 = 1000000000000
      -             0
        -------------
        1000000000000
0 → DC ← (1 crossed out)
```

* Overflow and underflow in the resulting mantissa can also affect the computed exponent.

BCD ARITHMETIC

TEN'S COMPLEMENT FOR BCD (CONT.)

If a mantissa of zero is complemented, the entire mantissa remains zero, and DC is not set, as you might expect. DC is <u>always</u> set to zero by CMX and CMY.

TEN'S COMPLEMENT ARITHMETIC DEMONSTRATION

Given n, subtract x, or, add -x: $S = n - x$ (1)
We assume only that the signs of n and x differ. The sign of S will be the same as n if $|n| > |x|$, and the same as x if $|x| > |n|$.

Complement x: $\overline{x} = 10^k - x$ (2)

($\Leftarrow$ denotes an operation we perform, but not necessarily strict mathematical equality)

Then: $S \Leftarrow (n + \overline{x}) \bmod 10^k = (n \div 10^k - x) \bmod 10^k$, or (3)

$$|S| \Leftarrow (|n| - |x| + 10^k) \bmod 10^k \quad (4)$$

Note: (Thing) mod 10^k is a way to denote the k right-most digits of an integer. We resort to this notational device because in a strict mathematical sense $s \neq n + 10^k - x$. (How can it, if S <u>really</u> equals n - x? There is a difference of 10^k!)

Line 4 is not as bad as it looks. First, it says that the k-digit sum is always formed as positive, regardles of its actual sign. Also, n and x are treated as positive, regardless of their signs. This is reminiscent of $|a - b| = ||a| - |b||$. Finally, a word about the k-digit restriction. It works because: a) to subtract, the firmware changes the sign of the subtrahend and proceeds as in addition; b) The complement mechanism is only used when addition involves opposing signs. Now, two k-digit things will have at most a k-digit difference.

<u>I Assume $|n| - |x| = d \geq 0$</u>

Then $S \Leftarrow (10^k + d) \bmod 10^k$

```
Now 10^K + d = <-----> 1000000  <- k zero's
                     +  ..d..   <- max of k digits
Overflow sets DC=1<-->1 ..d..   <- k + 1 digits total
```

Accordingly, we drop the overflow by simply noting that DC is set, and then ignoring (or perhaps clearing) it.

Thus, if overflow results, the resulting answer is the correct sequence of digits, and since $|n| \geq |x|$, the answer should be assigned the sign of n.

BCD ARITHMETIC

TEN'S COMPLEMENT ARITHMETIC DEMONSTRATION (CONT.)

II Assume $|n| - |x| = d < 0$ (note that $|S| = |d|$)

Then $S \Leftarrow (10^k - d) \bmod 10^k$

But $10^k - d = \bar{d}$, thus $S \Leftarrow \bar{d}$, but $|S| = |d|$, not $|\bar{d}|$

Note that 10^k - d already is at most k digits due to borrowing when doing the subtraction:

```
 0 9 9
 1 0 0 00000  <--- K ZERO'S
-   <-d-->    <--- MAX OF K DIGITS
---------
 0<---d̄--->
```

This guarantees that DC ends up a zero.

Thus, if the result in DC is zero, the answer needs to be re-complemented, and since $|n| < |x|$, the answer should be assigned the sign of x.

In the event we had choosen to complement n instead of x, the process would still work.

$S = n - x$

And $S \Leftarrow (10^k - |n| + |x|) \bmod 10^k$

But $10^k - |n| + |x| = 10^k - (|n| - |x|)$ and we have the same $|n| - |x|$ as before.

* * * * * * * * *

Here is the rule for doing decimal summations with ten's complements:

If the signs of the numbers are the same, simply add them and leave the signs alone.

BCD ARITHMETIC

TEN'S COMPLEMENT ARITHMETIC DEMONSTRATION (CONT.)

If the signs are different, complement one of the numbers, then add. If the result is accompanied by overflow, drop the overflow digit (DC). If overflow does not accompany the result, complement the answer. Ensure that the result is assigned the sign of the addend having the larger absolute value.

FLOATING-POINT SUMMATIONS

Specific procedures for implementing floating-point addition and subtraction vary widely. One thing that is fairly standard in this, however: To subtract, the software simply changes the sign of the subtrahend and proceeds as in addition. The addition routine is capable of handling all possibilities of signs and relative absolute values on two addends.

Another common practice is firmware checking of each addend for equality to zero. If either of the addends is zero, then the other addend is promptly taken as the answer.

OFFSETS

Addition can proceed only when the exponents of the two addends are the same. If they are not the same to start out with, they are made the same by shifting one of the mantissas an amount equal to the exponent difference.

This difference is easily found by subtracting the (algebraically) smaller exponent from the larger one. If the difference is eleven or less, it is possible to offset the mantissa of the number with the smaller exponent.

```
X.XXXXXXXXXXX E6 + Y.YYYYYYYYYYY E4
                                  ┌SAVED IN A
  X.X X X X X X X X X X X X  ↓  E6
+  .0 Y Y Y Y Y Y Y Y Y Y Y Y  E6
──────────────────────────────────
  Z.Z Z Z Z Z Z Z Z Z Z Z  ↑  E6
                           └THESE TWO DIGITS ARE LOST
                            DURING THE SHIFTING PROCESS,
                            EXCEPT FOR THE LEFT-MOST ONE,
                            WHICH IS SAVED IN A0-3 FOR
                            ROUND-OFF PURPOSES.
```

When offsetting mantissas for addition, the mantissa with the (algebraically) larger exponent is left alone, and mantissa with the (lagebraically) smaller exponent is the one that is right-shifted.

FLOATING-POINT SUMMATIONS

OFFSETS (CONT.)

As can be seen from the illustration, a shift of twelve or more digits would result in a mantissa of all zeros. The firmware detects the condition of an exponent difference greater than eleven, and simply takes the number with the larger exponent as the answer.

The EMC provides an n-many mantissa right-shift instruction for each of AR1 and AR2. These are MRX and MRY, respectively.

For each instruction, the number of digits to be shifted is assumed to be in the B register. Zero's are shifted into D_1*, and all but the last of the D_{12}'s is lost; it is saved in A, for round-off after the addition. Also, DC is set to zero in anticipation of the forthcoming addition activity.

MANTISSA ADDITION

The instruction FXA is used to add the mantissas after any necessary offset has been previously induced. FXA knows nothing of signs, complements, or exponents; it is strictly a positive-integer-addition process:

```
                    <AR1>=  D1 D2 D3 -------- D12
                    <AR2>=  D1 D2 D3 -------- D12
          +                                   <DC> <- INITIAL VALUE OF DC
         ______________________________________
(OVERFLOW) -> "D0"    D1 D2 D3 -------- D12  <- AR2
              ^
              |___DC (FINAL VALUE OF DC)
```

The reason for including DC itself in the addition of the D_{12}'s if that it would come in handy if FXA were used to add mantissas having more than 12 digits. In this way DC could function like the E register of the BPC.

If the signs of the original numbers were different, an overflow (DC=1) means that the resulting AR2 need not be complemented, and DC is to be ignored. Contriwise, a resulting DC of 0 means the resulting AR2 must be complemented, after which DC can be ignored.

* MRX and MRY do not necessarily shift in a zero on the first shift; on the first shift $<A_{0-3}>$ is what is shifted in, Subsequent shifts do shift in zero. During offsets in preparation for floating-point addition, the firmware ensures that $<A_{0-3}>$ = 0, however.

FLOATING-POINT SUMMATIONS

MANTISSA ADDITION (CONT.)

There are still some loose ends. Suppose the signs were the same, and DC ended up a 1? In such a case DC represents a digit of 1 to the left of D_1; AR2 plus DC constitute a 13 digits answer. What is required now is a one-digit right shift of AR2, shifting a 1 into D_1. MRY is the basis for this operation. Such a shift must also be accompanied by an increment (and test for overflow) of the AR2 exponent.

The situation described in the previous paragraph cannot occur if the original numbers had opposing signs. Why not??

The case of opposing signs has its own rub, however. Read on.

NORMALIZATION

The raw result of an arithmetic operation might not be a floating-point number that fits the standard form. It might have a leading DC needing to be incorporated into the number, as we have seen. Another possible deviation is a resulting D_1 of zero (and no overflow). There could also be several zero-digits as left-most digits of the mantissa.

Such a situation calls for the NRM instruction. It shifts AR2 left until D_1 is non-zero. The number of shifts is left as a binary number in the B register. The maximum number of shifts NRM will perform is 12. If NRM must do all 12 shifts, AR2 must have been zero. This is indicated by count of 12 in B, and well as by result of 1 in DC. For all other shift-counts, NRM leaves DC=0.

The firmware must complete the normalization process as follows:
The resulting number of shifts (in B) is subtracted from the AR2 exponent, and the result tested for underflow.

ROUNDING

The EMC does not have an instruction to automatically round a result - it is the firmware's responsibility to determine when to round, and there are various approaches to this problem. However, once the decision is made to round AR2 up (one count in D_{12}), the easiest way to do this is to set B to 000001_8, and execute an MWA.

This is in every respect the same as setting AR1 to one, and then doing an FXA, except that it is easier. Why not simply increment the word containing D_{12}? (D_{12} is on the far right of that word.) Such a move would not generate BCD carries if they were needed. If for instance, the mantissa being rounded up is all nines, the carry would need to propagate all the way up to DC.

ROUNDING (CONT.)

After rounding, AR2 must be checked for overflow, and if necessary, right-shifted with the exponent incremented and tested for overflow.

FLOATING-POINT MULTIPLICATION

This section will illustrate the function of the FMP instruction (fast multiply) as it is used in floating-point multiplications. We shall pursue this through the use of an example, assuming four-digit integers.

We can get by nicely on this because the exponents have only to do with the exponent of the preliminary answer (that is, possibly non-normal answer); the sequence of mantissa digits in the answer is determined solely by the digit-sequences of the multiplier and multiplicand. Therefore, we can treat the mantissas as integers during the actual multiply process.

The sign of the product is, of course, determined in advance by inspection of the signs of the original factors.

The fact that our illustration uses only four digits in no way invalidates the explanation; it merely reduces the amount of symbolism by eighty-nine percent.

Let's assume that the two mantissas we seek to multiply are:

Multiplicand = A B C D

Multiplier = W X Y Z

One symbolic way to indicate how this multiplication is done is:

$$
\begin{array}{lcccccccccl}
 & & & & & A & B & C & D & & \\
 & & & & (x) & W & X & Y & Z & & \\
\hline
(1) & & & & Z_{ov} & Z_1 & Z_2 & Z_3 & Z_4 & = & Z(ABCD)\times 10^0 \\
(2) & & & Y_{ov} & Y_1 & Y_2 & Y_3 & Y_4 & 0 & = & Y(ABCD)\times 10^1 \\
(3) & & X_{ov} & X_1 & X_2 & X_3 & X_4 & 0 & 0 & = & X(ABCD)\times 10^2 \\
(4)\ + & W_{ov} & W_1 & W_2 & W_3 & W_4 & 0 & 0 & 0 & = & W(ABCD)\times 10^3 \\
\hline
\end{array}
$$

[—EIGHT DIGIT NUMBER—]

FLOATING-POINT MULTIPLICATION (CONT.)

Consider how Z_{OV} Z_1 Z_2 Z_3 Z_4 is found (this is where FMP is used). It is really ABCD added to itself Z-times. Similarly, Y_{OV} Y_1 Y_2 Y_3 Y_4 is ABCD added to itself Y-times. Prior to adding line 1 to line 2, we shift line 1 one digit to the right (including Z_{OV}-it goes into the new Z_1). This allows line 2 to have ten times the weight of line 1. The resulting summation is shifted once to the right and added to line 3, and so on. These shifts are illustrated by the right-most zeros in lines 2, 3 and 4.

Now lets take a moment and look at how FMP generates a partial product. Consider Z(ABCD). AR2 is cleared and AR1 loaded with ABCD. B_{0-3} contains Z. Now FMP is given. AR1 and AR2 are added together Z-times, producing Z(ABCD) in AR2. The digit Z_{OV} ends up in A_{0-3}. It can be anything from a zero to an eight.* Notice that the mantissa right-shifts MRX and MRY each shift $<A_{0-3}>$ into D_1. So the right-shifting of the partial product also takes care of retaining its overflow digit.

Now we are ready to find Y_{OV} Y_1 Y_2 Y_3 Y_4. Generally speaking, this is not found separately and then added to Z_{OV} Z_1 Z_2 Z_3. Instead, ABCD is merely added to Z_{OV} Z_1 Z_2 Z_3 Y-times. This both increases speed and saves memory over saving all partial products before summation, with no undue loss of accuracy. As before, the overflow digit Y_{OV} is left in A_{0-3}. And so it goes, AR2 is shifted right one more time, making Y_{OV} the left-most digit of the partial products as summed to date. B_{0-3} is made to contain X, and FMP is given a third time.

We can make a number of minor points in conclusion. First, at each step of partial product summation we throw away a significant digit due to the shift. This can't be helped. In general, the product of two 12 digits numbers has 24 digits of precision, but we are limited to 12, so we throw the bottom 12 digits away.*

These digits can be inspected, however. The MRY used to shift AR2 puts the lost digit into A_{0-3}. This provides an easy way for a rounding mechanism to check on those digits as they tossed out. Indeed, the rounding routine will need to save the last digit thrown out, for use in rounding in the event the last use of FMP produces no overflow digit.

Lastly, notice that we can put WXYZ into B at the very start of the process, and simply shift B right with and SBR 4 in-between uses of FMP. After all, FMP uses only $< B_{0-3} >$ as the number of times to add AR1 to AR2.

* When adding ABCD to ABCD, the worst carry that can occur is a 1 preceeding a remaining four digits of sum. For each subsequent add of ABCD to the sum, the left-most digit can only increase by one. But to multiply a number by nine (the worst case), you only add it to itself with eight additions. Hence a maximum of eight for the overflow digit.

* An error analysis of this algorithm discloses that dropping these digits causes the answer, <u>on the average</u>, to be slightly smaller than it should be. Rounding introduces a similar error in the other direction.

FLOATING-POINT BCD DIVISION

So much for the easy part. The other arithmetic operations could be explained without too much ado, and their pertinent EMC instructions readily rationalized in terms of the desired activity. Not only is the floating-point division algorithm messier and inherently less obvious, but we shall have to resort to examining a section of code to get a clear idea of how FDV is actually employed. This is necessary because FDV does not, percentage-wise, do as much for division as, say, FMP does for multiply (← author's opinion).

THE DIVISION ALGORITHM

Somebody out there is probably muttering: "Wait a minute, why can't they just reverse the multiplication process....?" The answer is "significant digits". Suppose a 12-digit DVD had been found by multiplying Q by DVR, each of which were 12-digit numbers (then Q = DVD/DVR). The multiplication would have produced a 24-digit DVD; but we throw the least-significant 12 digits away. In order to reverse the multiplication process we would have to have those missing digits. But divide only ever has 12-digit numbers to work with. So a different procedure is needed. We take the coward's way out, and choose one that is essentially the same as the pencil and paper method for long division.

As in multiplication, the sign and exponent of the intermediate answer can be determined in advance.

Suppose we are going to divide:

(1) $480/15 = 32$

(2) THEN $(32)\cdot(15) = 480$

(3) $(32)\cdot(15) = (30+2)\cdot(15) = (30)\cdot(15) + (2)\cdot(15)$

(4) $= (3)\cdot(150) + (2)\cdot(15)$

We want to do this thing as a series of subtractions. However, we resist the folly of subtracting 15 from 480 thirty-two times! Instead, we look at line (4), and note that there are three 150's in 480. Perhaps if we subtracted them out and then found out how many 15's were in the difference.....Yes!

If you did that, you'd find that indeed, 150 can be subtracted three times, leaving a remainder of 30, and that 15 can be subtracted from 30 two times. Now, since subtracting 150 three times is the same as subtracting 15 thirty times (after all, 150 × 3 = 15 × 30), there must be (30 + 2) 15's in 480. So the answer is 32.

The division algorithm we are going to develop uses just a scheme. Following are some points to keep in mind.

FLOATING-POINT BCD DIVISION

THE DIVISION ALGORITHM (CONT.)

The digit sequence of the quotient is determined solely by the digit sequences of the mantissas of the dividend and the divisor - the mantissas are always normalized to begin with, and the exponents do not enter into the actual division activity. Thus our above example illustrates (in a three digit machine) the division of any number whose mantissa is 4.80 by any other number whose mantissa is 1.50:

$$4.80 \times 10^3 / 1.5 \times 10^{-2} = 3.20 \times 10^5$$

Just as for the previous operations we have examined, the easiest way is to forget about the alleged decimal point between D_1 and D_2, and consider the mantissas to be 12-digit integers.

The divisor will be in AR1 (memory outside the EMC) and the dividend in AR2 (accumulator registers with the EMC). The basic activity is to subtract AR1 from AR2 until AR2 gets smaller than AR1. The number of subtractions required for that to occur is the next digit of the quotient. Then AR2 is shifted left and the process is repeated until either a zero remainder occurs, or sufficient digits have been calculated, whichever occurs first. The quotient digits are merged, one at a time, into a complete quotient held in R/W memory. This is the firmware's responsibility, and it alone determines where in R/W the quotient is kept.

Now:

1) D_1 of the quotient might be zero (suppose AR1 is greater than the original AR2). In that case we accept the zero and shift as described below.

2) The number of subtractions will always be nine or fewer. This is because D_1 of AR1 can't be zero. You may want to think about that a minute and convince yourself.

3) If (1) occurs, or, after successful application of (2), we need to do something that corresponds to changing the 150 to 15 and getting ready to subtract it from 30 (the remainder).

   Now for various reasons we don't want to fool around with the 150. Instead, we shift the 30 left and make it 300. We get the same result, however.

4) If (1) occurs for D_1 of the quotient, it can't also occur for D_2. The basic reason for this is that D_1 of AR2 can't initially be zero. After D_1, "zero" quotient-digits can occur for several digits in a row, however. But because 00--- can't occur, it is always sufficient to compute 13 digits (assuming no extra digit for rounding - and counting a leading zero as one of the 13).*

* Suppose the leading quotient digit were zero. Then you might consider computing 14 digits, so that after normalization (when there would only be 13 digits left) you would be able to round to 12 digits based on the 13th digit. That sample division routine given shortly does not do this.

FLOATING-POINT BCD DIVISION

THE DIVISION ALGORITHM (CONT.)

5) Consider a (1)-like situation for either D_1 or some other digit of the quotient. The necessary shift (via MLY) moves the left-most digit of AR2 into A. We cannot ignore this digit when subtracting AR1. Indeed, now we must deal with a 13-digit dividend; A followed by AR2. Here is some bad news; FDV knows nothing of 13 digit arithmetic!! The software's use of FDV will have to make up the difference.

THE FDV INSTRUCTION

FDV is used to accomplish the equivalent of automatically repeated subtraction of AR1 from AR2, until AR2 becomes smaller than AR1. It does this by adding AR1 to AR2 until overflow occurs. This assumes that AR2 has been complemented prior to the execution of FDV.

Your author feels that it makes more sense to describe floating-point division in terms of subtractions, rather than additions to a complement. We shall designate subtractions that are really complement-additions as "subtractions".

FDV returns the number of successful "subtractions" as a binary number (same as BCD) in B_{0-3}; B_{4-15} are returned as zero.

In general, after an application of FDV it is necessary to patch-up AR2 before shifting and using FDV again. This is because AR2 retains the result of the first *unsuccessful* "subtraction". What is done is to de-complement AR2 and add AR1 back one time, so as to undo the effect of the unsuccessful "subtraction". Then AR2 is shifted, and then complemented. AR1 remains untouched throughout the entire process.

There is one case where AR2 does not need to be adjusted. This is when the result in AR2 is zero. This means that the divisor is contained within the dividend exactly an integral number of times. This produces an eventual zero remainder (the result in AR2). We say that such an event generates a *perfect quotient*.

Now, in the event of a perfect quotient the number returned in B_{0-3} is one count too small. (You might have to think about that for a few minutes - but its true. Normally, overflow is associated with the first unsuccessful "subtraction" because the answer should really be negative. But it just so happens that the generation of a result of zero - which is basically still a successful "subtraction" - is accompanied by overflow.) So the loop that employs FDV has constantly got to be on the look-out for a perfect quotient. This is desirable for another reason. Once a perfect quotient has been discovered, it is undesirable to proceed with further division activity.

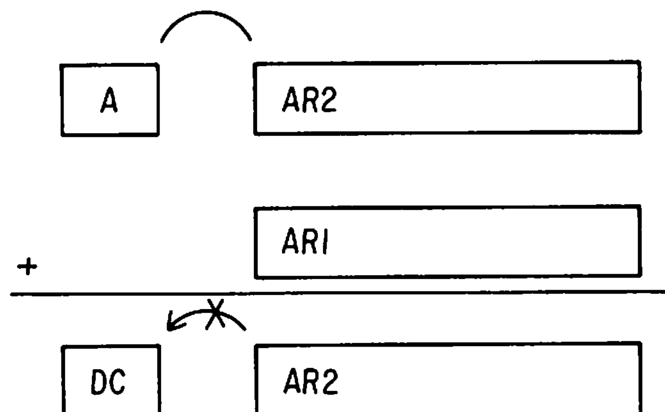
THE FDV INSTRUCTION (CONT.)

Another aspect of FDV to be aware of is the way it returns quotient digits into B. Each digit is placed into B_{0-3}, and B_{4-15} *are cleared*. This means you can't simply shift B left in-between the extraction of four consecutive quotient digits, and then store B into the sequence of words used to receive the answer. Instead, the sequence of digits has to be individually stored in the answer as they are found; B cannot be used as temporary storage for a group of quotient digits.*

There is one last fly in the ointment. This is the business of the dividend frequently being 13 digits; A followed by AR2. Your author knows of only one solution to this, and it's a good one, but it will take some explaining. Clever things tend to not be obvious.

A series of FDV's can be used to "subtract" a 12-digit AR1 from a 13-digit A-followed-by-AR2.

Suppose we have a complemented 13-digit number in A and AR2, as shown below:



When FDV is given it adds the 12 digits of AR1 and the 12 digits of AR2 together until an overflow occurs. (FDV does *not* set DC, however.) Now if FDV were a 13-digit operation the carry from AR2 would be used to increment A. Also, there is nothing wrong with the resulting digit sequence in AR2. The digits simply "turn-over" and keep going. But after each FDV the software has to "increment A and detect when it goes from nine to ten".* When the digit in A goes from nine to ten we have "real overflow" of the 13 digit number.

* This drawback would be avoided if FDV simply returned the number of successful "subtractions" to B_{0-3}, leaving B_{4-15} entirely alone. The designers of the EMC were well aware of this, but faced internal constraints, such as chip size, and number of internal states. These constraints prevented the implementation of the more desirable definition.

* That, or equivalent behavior. The example we develop later doesn't physically do exactly what's shown above - but what it does do is equivalent to it.

THE FDV INSTRUCTION (CONT.)

Each use of FDV adds ARI and AR2 (into AR2) until AR2 overflows. When that happens we increment A and add again with FDV if A is less than ten - no adjustment is made to the digit sequence in AR2 - none is needed. *But, the digit sequence of AR2 reflects the "subtraction" that produces the overflow. The number returned to B is one less than that.* AR2 and B_{0-3} are out of step, so to speak.

What we want to know is the total number of possible "subtractions" of ARI from A-AR2. We get that number by summing the values of < B > + I for all uses of FDV, except the last one, during the 12-from-13-digit "subtractions". The resulting digit sequence in AR2, when the 12-from-13-digit-"subtraction" is *completed*, is like always, the result of an overflow, which in this case we don't want. So as before if there is no perfect quotient, AR2 will be de-complemented and ARI added to it. Then the previous FDV needs to contribute only $< B_{0-3} >$ to the sum of the latest quotient digit, not $< B_{0-3} > + 1$.

For example, if there were three uses of FDV for a certain quotient digit of a 12-from-13-digit "subtraction", we would form the (non-perfect) quotient digit as:

$$Q_n = (< B_{0-3} > + 1) + (< B_{0-3} > + 1) + < B_{0-3} >$$

AFTER Ist USE OF FDV — AFTER 2nd USE OF FDV — AFTER FINAL USE OF FDV

If the same general situation produced a perfect quotient on the nth digit, then for the same reasons as before, we do not count the last "subtraction":

$$Q_n = (< B_{0-3} > + 1) + (< B_{0-3} > + 1) + (< B_{0-3} > + 1)$$

AFTER Ist USE OF FDV — AFTER 2nd USE OF FDV — AFTER FINAL USE OF FDV

Somebody is probably wondering what happens if Q_n turns out to be greater than nine. It doesn't. Ever. Think in terms of the uncomplemented 13 digit A-AR2. That number is always less than ten times greater than ARI (D_1 of ARI $\neq 0$, remember). This is left as an exercise for the reader - it's not worth pursuing here.

As a matter of implementation, it is tedious to check if A has been incremented to ten. We can always tell in advance, from each new and uncomplemented value that is shifted into A, how many overflows out of AR2

THE FDV INSTRUCTION (CONT.)

would be required if we *were* to increment and test on A. The easiest thing to do is to put that number of needed FDV's into A as a count to be either incremented or decremented to zero. Then each use of FDV for a 12-from-13 digit subtraction updates A until A is zero.

In the sample program segment that follows, the value returned to B_{0-3} is always incremented by one immediately after it is returned. The increment will later be taken out as the quotient digit is stored in its final destination,* *provided that it should be taken out.* It is easier to always do the increment and then test for when to take it out, rather than to test for when to put it in.

SAMPLE DIVISION ROUTINE

The rule is this:

1) Always increment the value returned in B_{0-3}.
2) First check for multiple FDV's as a part of a 12-from-13-digit subtraction. If so, loop immediately, performing no other tests or activities.
3) When a quotient digit has been found, check to see if the quotient is now a perfect quotient. If so, exit the division loop without removing the last increment. Save the last digit found as part of the answer.
4) If the quotient is not a perfect quotient, decrement the value of the last quotient digit found, and save it as part of the answer.

The test for a perfect quotient is simple, although not super-short: if AR2 is zero the divisor has subtracted out evenly from the dividend.

The sample segment shown does not include the testing for and handling of these things:

1) signs
2) division by zero
3) division into zero
4) exponents
5) overflow
6) rounding

All of these areas are handled by additional code segments not part of the division loop proper.

* Final destination here means with respect to the divide routine, and is probably a temporary location, not the final destination called for in the user's program.

```
0001  *
0002  * USEFUL EQUATES
0003  *
0004  AR2M1 EQU AR2+1    (=21B) #1 AR2 MANTISSA WORD
0005  AR2M2 EQU AR2+2    (=22B) #2 AR2 MANTISSA WORD
0006  AR2M3 EQU AR2+3    (=23B) #3 AR2 MANTISSA WORD
0007        .
0008        .
0009        .
0010        .
0011        .
0012        .
0013  *
0014  * THESE WORDS IN ROM
0015  *
0016  M10D  DEC -10
0017  M1D   DEC -1
0018  ZERO  OCT 0
0019  P1D   DEC 1
0020  P4D   DEC 4
0021  P13D  DEC 13
0022  P17B  OCT 17
0023  P20B  OCT 20
0024  QWPIV DEF QW1-1   PERMANENT STARTING VALUE OF QWPTR
0025        .
0026        .
0027        .
0028        .
0029        .
0030        .
0031  *
0032  * THESE WORDS IN READ/WRITE
0033  *
0034  QWPTR BSS 1   QUOTIENT WORD POINTER
0035  QW1   BSS 1   QUOTIENT WORD #1
0036  QW2   BSS 1   QUOTIENT WORD #2
0037  QW3   BSS 1   QUOTIENT WORD #3
0038  QW4   BSS 1   QUOTIENT WORD #4 (FOR DIGIT #13)
0039  DIGCT BSS 1   DIGIT COUNTER (13 - 1)
0040  WWDCT BSS 1   WITHIN WORD DIGIT COUNTER  (1 - 4)
0041  FDVCT BSS 1   FDV RE-APPLICATION COUNTER
0042        .
0043        .
0044        .
0045        .
0046        .
0047        .
0048  *
0049  * DIVIDEND ALREADY IN AR2
0050  * DIVISOR ALREADY IN AR1
0051  * START OF FUNDAMENTAL DIVISION LOOP
0052  *
0053  DIVID LDA QWPIV   SET QUOTIENT WORD POINTER TO
0054        STA QWPTR     INITIAL VALUE (=QW1-1)
0055        CMY         COMPLEMENT THE DIVIDEND
0056        LDB P13D    (=+13 DEC)
0057        STB DIGCT   INITIALIZE DIGIT COUNT TO 13
0058        LDA M1D     (=-1 DEC) INITIALIZE FDV REP COUNT FOR DIGIT #1
0059  *
0060  DNXTW ISZ QWPTR   INCREMENT QUOTIENT WORD POINTER
```

```
0061        LDB P4D     (=+4 DEC) SET THE WITHIN-WORD
0062        STB WWDCT       COUNT TO 4
0063  *
0064  DNXTD SBL 4       CLEAR B<0-3>
0065        STB QWPTR,I CLEAR NEXT WORD IN RECIEVING LOCATION
0066        STA FDVCT   STORE NEXT DIGIT FDV REP COUNT
0067  *
0068  FDVLP FDV         AR2=AR2+AR1 UNTIL OVERFLOW
0069        ADB QWPTR,I MERGE NEW DIGIT WITH REST OF CURRENT ANSWER WORD
0070        ADB P1D     INCREMENT THE NEW DIGIT
0071        STB QWPTR,I SAVE THIS NEWEST PIECE OF THE ANSWER
0072  *
0073        ISZ FDVCT   INCREMENT FDV REP COUNT, LOOP IF NON-ZERO
0074        JMP FDVLP   UNFINISHED 12-FROM-13-DIGIT SUBTRACTION, RE-DO FDV
0075  *
0076        LDA AR2M1   "OR" ALL 3 WORDS OF THE AR2 MANTISSA
0077        IOR AR2M2     TOGETHER. CHECK FOR RESULTING ALL
0078        IOR AR2M3     ZEROS. IF SO, THEN HAVE
0079        SZA YESPQ     PERFECT QUOTIENT.
0080  *
0081  * NO PERFECT QUOTIENT. DIVIDE AGAIN, BUT FIRST RESTORE DIVIDEND,
0082  * SHIFT IT LEFT, AND THEN FIND NEW FDV REP COUNT.
0083  *
0084        CMY         DECOMPLEMENT REMAINDER (AR2)
0085        FXA         ADD BACK DIVISOR (AR1)
0086        LDB QWPTR,I GET LAST CALCULATED DIGIT
0087        ADB M1D     UNDO LATEST (AND UN-NEEDED) INCREMENT
0088        STB QWPTR,I SAVE THE NOW CORRECT PARTIAL ANSWER
0089        CMY         COMPLEMENT NEW DIVIDEND (AR2)
0090  *
0091        LDA ZERO    CLEAR A SO AS TO NOT SHIFT IN JUNK BELOW
0092        MLY         SHIFT DIVIDEND LEFT
0093        ADA M10D    FIND NEXT FDV REP COUNT
0094  *
0095  * THE FDV REP COUNT IN A IS NEGATIVE SO THAT IT CAN BE COUNTED
0096  * UP TO ZERO. THE ABSOLUTE VALUE OF A IS THE NUMBER OF TIMES
0097  * FDV WILL BE APPLIED FOR THE QUOTIENT DIGIT BEING FOUND. FOR
0098  * A 12-DIGIT-FROM-12-DIGIT-SUBTRACTION, A=-1, AS ONLY ONE USE
0099  * OF FDV IS REQUIRED.
0100  *
0101  * THE MLY SHIFTS INTO THE A-REG A DIGIT WHOSE VALUE IS 9-D1
0102  * WITH RESPECT TO THE UNCOMPLEMENTED AR2 (PRIOR TO ITS SHIFT).
0103  * NOW, 9-D1-10 IS SIMPLY -(D1+1). FORGETTING THE MINUS SIGN FOR
0104  * A MOMENT, THIS SAYS THAT THE A-REG IS ONE COUNT HIGHER THAN
0105  * THE "REAL" LEFT-MOST DIGIT OF THE DIVIDEND. REMEMBERING THAT
0106  * A IS INCREMENTED UP TO ZERO, IF THE "REAL" DIGIT IS ZERO, THEN
0107  * ONE FDV IS DONE. IF THE "REAL" LEFT-MOST DIGIT IS ONE, THEN AN
0108  * EXTRA FDV IS DONE. FOR TWO, THREE FDV'S, ETC., ETC.
0109  *
0110  *
0111  * BOTTOM-OF-LOOP MAINTENANCE FOLLOWS
0112  *
0113        DSZ DIGCT   DECREMENT TOTAL DIGIT COUNT, DONE IF ZERO
0114        JMP *+2     NOT DONE, DIVIDE SOME MORE
0115        JMP DONE    GO FINISH UP
0116        DSZ WWDCT   DECREMENT WITHIN-WORD DIGIT COUNT
0117        JMP DNXTD   LOOP FOR NEXT DIGIT WITHIN SAME QUOTIENT WORD
0118        JMP DNXTW   LOOP FOR NEXT DIGIT IN NEXT QUOTIENT WORD
0119  *
0120  YESPQ DSZ DIGCT   PERFECT QUOTIENT BEFORE ALL 13 DIGITS FOUND?
```

```
0121        JMP YES
0122        JMP DONE    NO, PERFECT QUOTIENT ON DIGIT #13
0123  *
0124        SBL 4
0125  YES   DSZ WWDCT   SHIFT LATEST DIGITS TO LEFT AS NECESSARY
0126        JMP *-2
0127  *
0128  DONE  STB QWPTR,I STORE LAST DIGITS OF QUOTIENT
0129        LDA QWPIV   SET "FROM" X-FER ADDRESS
0130        ADA P1D
0131        LDB P20B    SET "TO" X-FER ADDRESS
0132        XFR 4       X-FER QUOTIENT TO AR2
0133  *
0134        NRM         NORMALIZE THE QUOTIENT IF NEEDED
0135        SZB GO.ON   GO ON IF IT WAS ALREADY OK, JOE
0136  *
0137  * HERE, THE FIRST DIGIT OF THE QUOTIENT WAS A ZERO. NRM GOT RID
0138  * OF THAT AND NOW WE PUT THE OLD DIGIT #13 IN AS THE NEW DIGIT #12.
0139  *
0140        LDA QW4     GET DIGIT #13
0141        AND P17B    RESTRICT IT TO 4 BITS
0142  * ABOVE INST NEEDED ONLY IF QW4 USED ELSEWHERE FOR OTHER THINGS
0143        ADA QW3     PUT IT IN AS NEW DIGIT #12 (OLD DIGIT #12=0)
0144        STA QW3     RESTORE THIRD WORD OF QUOTIENT
0145        LDB         SET EXPONENT ADJUST FLAG
0146         .
0147         .
0148         .
0149         .
0150         .
0151         .
0152         .
0153  GO.ON .............
0154         .
0155         .
0156         .
0157         .
0158         .
0159         .
```

INTRODUCTION TO THE MACHINE INSTRUCTIONS

NOTATION

Assembly language machine instructions are three-letter mnemonics. Each machine instruction source statement corresponds to a machine-operation in the object program produced by the assembler. Notation used is representing source statements is explained below:

label	Optional statement label. Labels must begin with an alphabetic character, period, or certain other non-numeric characters. Labels may be one through five characters in length. If present, a label must begin in column 1. A space terminates a label. If a statement does not have label, then column 1 must be a blank.
m	Memory location. This can be an octal or decimal integer, a symbol used as a label elsewhere, or, an expression composed of a combination of these combined through + and - operators. Parentheses are not permitted in expressions.
n (lower case)	Numerical quantity. A numeric value that is not an address, but represents a shift or skip amount.
N (upper case)	Octal or decimal constant whose value is restricted to the range: $1 \leq N \leq 20_8 = 16_{10}$ ASMA allows N to also be any expression, provided that the value of the expression is within the stated range.
I	Indirect addressing indicator for memory reference instructions. Also indicates an automatic increment for place and withdraw instructions.
D	Decrement indicator for place and withdraw instructions.
P	Indicator used in Return instructions to instruct the IOC to pop its peripheral address stack.
reg. 0-7	Register location. This can be an octal or decimal integer, or an assembler-pre-defined symbol. It might even be an expression. Regardless of what it is, it must have a value of 0_8 through 7_8, inclusive.
reg. 4-7	Register location. Same rules as for reg. 0-7 above, except the value must be 4_8 - 7_8, inclusive.
.../...	The slash indicates the item on either side (but not both) may be used at this place in the source statement.
comments	Optional comments. Comments must be separated by at least one space from the material to the left of the comment.
[ ]	Brackets indicate that the item contained within them is optional.

BPC MACHINE INSTRUCTIONS

MEMORY REFERENCE GROUP

Each of the 14 memory reference instructions performs some operation based upon the contents of a referenced memory location. Unless the reference is to a location on the base page, it must be on the same current page as the instruction. The assembler determines which type of page-reference is used, and sets the B/C bit (bit 10) of the instruction accordingly. The least ten significant bits of the address of the referenced location are enclosed in bits 0-9 of the instruction. A memory reference may be indirect. In the source this is indicated with a ,I after the operand. This is assembled by making bit 15 of the instruction be a one.

label	LDA	m [,I]	comments

Load A from m. The A register is loaded with the contents of the addressed memory location.

label	LDB	m [,I]	comments

Load B from m. The B register is loaded with the contents of the addressed memory location.

label	CPA	m [,I]	comments

Compare the contents of m with the contents of A; skip if unequal. The two 16-bit words are compared bit by bit. If they differ the next instruction is skipped, otherwise it is executed next.

label	CPB	m [,I]	comments

Compare the contents of m with the contents of B; skip if unequal. The two 16-bit words are compared bit by bit. If they differ the next instruction is skipped, otherwise it is executed next.

label	ADA	m [,I]	comments

Add the contents of m to A. The contents of the addressed memory location are added to those of A. The binary sum remains in A while the contents of m remain unchanged. If a carry occurs from bit 15 the E register is set to a one, otherwise, E is left unchanged. If an overflow occurs the OV register is set to a one, otherwise the OV register is left unchanged. The overflow condition occurs if there is a carry from either bits 14 or 15, but not both together.

label	ADB	m [,I]	comments

Add the contents of m to B. Otherwise identical to ADA.

label	STA	m [,I]	comments

Store the contents of A in m. The contents of the A register are stored into the addressed memory location, whose previous contents are lost.

MEMORY REFERENCE GROUP (CONT.)

label	STB	m [,I]	comments

Store the contents of B in m. The contents of the B register are stored into the addressed memory location, whose previous contents are lost.

label	JSM	m [,I]	comments

Jump to subroutine. JSM permits jumping to subroutines in either ROM or R/W memory. The value of the pointer in the return stack register (R) is incremented by one and the value of P (the location of the JSM) is stored in R,I. Program execution resumes at m.

label	JMP	m [,I]	comments

Jump to m. Program execution continues at location m.

label	ISZ	m [,I]	comments

Increment m; skip if zero. ISZ adds one to the contents of the referenced location, and writes the sum into that location. If the sum is zero, the next instruction is skipped. ISZ does not alter the contents of E and OV.

label	DSZ	m [,I]	comments

Decrement m, skip if zero. DSZ subtracts one from the contents of the referenced location, and writes the difference into that location. If the difference is zero, the next instruction is skipped. DSZ does not alter the contents of E and OV.

label	AND	m [,I]	comments

Logical and of A and m. The contents of A and m are and'ed, bit by bit, and the result is left in A.

label	IOR	m [,I]	comments

Inclusive or of A and m. The contents of A and m are or'ed, bit by bit, and the result is left in A. The inclusive or is the "ordinary or" operation.

The following four instructions are not, in the strictest sense, memory reference instructions. They are included here for the sake of continuity.

label	RET	m [,P]	comments

Return. The R register is a pointer into a stack of words in R/W memory containing the addresses of previous subroutine calls. A read R,I occurs. That produces the address (P) of the latest JSM that occurred. The BPC then jumps to address P+n, and R is decremented. The value of n may range from -32 to 31, inclusive. The value of n is encoded into bits 0 through 5 of the instructions as a 6 bit, two's complement, binary number.

MEMORY REFERENCE GROUP (CONT.)

The ordinary, everyday garden variety return is RET 1.

If a P is present, it "pops" the interrupt system. Two things occur when this happens: first, the peripheral address stack is popped, and second, the interrupt grant network is "decremented".

The peripheral address stack is a genuine hardware stack, 4 bits wide, and three levels deep. On the top of this stack is the current select code for I/O operations. Select codes are stacked as interrupts occur during I/O operations - A RET 0, P at the end of an interrupt service routine puts the select code of the interrupted device back on the top of the stack.

The interrupt grant network keeps track of which interrupt priority level is currently in use. From this it determines whether or not to grant an interrupt request. A RET 0, P at the end of an interrupt service routine causes the interrupt grant network to change the current interrupt priority level to the next lower level (unless it is already at the lowest level).

label	CLA	comments

Clear A. There is no machine-instruction called Clear A. The assembler turns this mnemonic into an SAR 16 (shift A right 16). This has the effect of clearing the A register.*

label	CLB	comments

Clear B. There is no machine-instruction called Clear B. The assembler turns this mnemonic into an SBR 16 (shift B right 16). This has the effect of clearing the B register.*

label	NOP	comments

Null operation. There is no machine-instruction for a no-operation, per se. The assembler turns this mnemonic into a LDA A, (the machine-instruction for which happens to be all zeros).

SHIFT-ROTATE GROUP

The shift-rotate instructions perform re-arrangements of the bits of the A and B registers. Each shift-rotate instruction includes a four-bit field in which the shift or rotate amount is encoded. The number to be encoded in the field is represented by n. In the source text n may range from 1 to 16, inclusive. The four-bit field (bits 0 through 3) will contain the binary code for n-1.

* CLA and CLB are probably not the best way to accomplish the desired result. If the program has in it a word that is all zeros, then it is faster to LDA or LDB with that word.

SHIFT-ROTATE GROUP (CONT.)

label	AAR	n	comments

Arithmetic right shift of A. The A register is shifted right n places with the sign bit (bit 15) filling all vacated bit positions; the n+1 most significant bits become equal to the sign bit.

label	ABR	n	comments

Arithmetic right shift of B. The B register is shifted right n places with the sign bit (bit 15) filling all vacated bit positions; the n+1 most significant bits become equal to the sign bit.

label	SAR	n	comments

Shift A right. The A register is shifted right n places with all vacated bit positions cleared; the n most significant bits become zeros.

label	SBR	n	comments

Shift B right. The B register is shifted right n places with all vacated bit positions cleared; the n most significant bits become zeros.

label	SAL	n	comments

Shift A left. The A register is shifted left n places; the n least significant bits become zeros.

label	SBL	n	comments

Shift B left. The B register is shifted left n places; the n least significant bits become zeros.

label	RAR	n	comments

Rotate A right. The A register is rotated right n places, with bit 0 rotating into bit 15.

label	RBR	n	comments

Rotate B right. The B register is rotated right n places, with bit 0 rotating into bit 15.

ALTER-SKIP GROUP

The alter-skip instructions each contain a six bit field which allows a relative branch of any of 64 locations. The distance of the branch is represented by a displacement, n; n may be within the range of -32 to 31, inclusive.

ALTER-SKIP GROUP (CONT.)

The arguments for the instructions of this group are shown as *±n, or, m. An argument of n by itself will generally cause an error. Internally, the assembler subtracts the current value of * from the argument as part of the evaluation process. So *±n-* is simply ±n, and m-* becomes a relative displacement rather than an actual address. This business of subtracting * was done to allow symbols and addresses (these are m's) as arguments. Thus it is possible to write SZA HOOK. All that is required is that HOOK be within the allowable skip distance of the instruction.

Bits 0 through 5 are coded with the value of n (or m-*) as follows: if the value is positive or zero, bit 5 is zero, and bits 0 through 4 receive the straight binary code for the value of n - if the value is negative, bit 5 is a 1, and bits 0 through 4 receive a complemented and incremented binary code.

For n or m-* =	bits 5 - 0	meaning:
-32	100000	if skip, next instruction is *-32
- 7	111001	if skip, next instruction is *-7
- 1	111111	if skip, next instruction is *-1
0	000000	if skip, repeat this instruction
1	000001	do next instruction, regardless
7	000111	if skip, next instruction is *+7
31	011111	if skip, next instruction is *+31

All instructions in the alter-skip group have the "skip" properties outlined above. Some of the instructions also have an optional "alter" property. This is where the general instruction form "skip if <some one bit condition>" is supplemented with the ability to alter the state of the bit mentioned in the condition. The alteration is to either set the bit, or clear it. If specified, the alteration is done *after* the condition is tested, never before.

To indicate in a source statement that an instruction includes the alter option, and to specify whether to clear or to set the tested bit, a comma-C or comma-S follows *±n/m. The C indicates clearing the bit, while an S indicates setting the bit.

The "alter" information is encoded into the 16 bit instruction word with 2 bits. For such instructions, bit 7 is called the $H/\overline{H}$ (Hold/Don't Hold) bit, and bit 6 is the C/S (Clear/Set) bit. If bit 7 is a zero (specifying H) the "alter" option is not active; neither S nor C followed n in the source statement of the instruction, and the tested bit is left unchanged. If bit 7 is a 1 (specifying $\overline{H}$), then "alter" option is active, and bit 6 specifies whether it is S or C.

label	SZA	* ± n/m	comments

Skip if A zero. If all 16 bits of the A register are zero, skip the amount indicated by n, or, to m.

ALTER-SKIP GROUP (CONT.)

label	SZB	* ± n/m	comments

Skip if B zero. If all 16 bits of the B register are zero, skip the amount indicated by n, or, to m.

label	RZA	* ± n/m	comments

Skip if A not zero. If any of the 16 bits of the A register are set, skip the amount indicated by n, or, to m.

label	RZB	* ± n/m	comments

Skip if B not zero. If any of the 16 bits of the B register are set, skip the amount indicated by n, or, to m.

label	SIA	* ± n/m	comments

Skip if A zero, and then increment A. The A register is tested, and then incremented by one. If all 16 bits of A were zero before the increment, skip the amount indicated by n, or, to m. SIA does not affect the contents of E or OV.

label	SIB	* ± n/m	comments

Skip if B zero, and then increment B. The B register is tested, and then incremented by one. If all 16 bits of B were zero before the increment, skip the amount indicated by n, or, to m. SIB does not affect the contents of E or OV.

label	RIA	* ± n/m	comments

Skip if A not zero, and then increment A. The A register is tested, and then incremented by one. If any bits of A were one before the increment, skip the amount indicated by n, or, to m. RIA does not affect the contents of E or OV.

label	RIB	* ± n/m	comments

Skip if B not zero, and then increment B. The B register is tested, and then incremented by one. If any bits of B were one before the increment, skip the amount indicated by n, or, to m. RIB does not affect the contents of E or OV.

In connection with the next four instructions, Flag and Status are controlled by the peripheral interface addressed by the current select code. The select code is the number that is stored in the register named PA, located in the IOC. Both Status and Flag originate such that when a missing interface is addressed Status and Flag will appear to be false, or not set.

ALTER-SKIP GROUP (CONT.)

label	SFS	* ± n/m	comments

Skip if Flag line set. If the Flag line is true, skip the amount indicated by n, or, to m.

label	SFC	* ± n/m	comments

Skip if Flag line clear. If the Flag line is false, skip the amount indicated by n, or, to m.

label	SSS	* ± n/m	comments

Skip if Status line set. If the Status line is true, skip the amount indicated by n, or, to m.

label	SSC	* ± n/m	comments

Skip if Status line clear. If the Status line is false, skip the amount indicated by n, or, to m.

label	SDS	* ± n/m	comments

Skip if Decimal Carry set. Decimal Carry (DC) is a one-bit register in the EMC. It is controlled by the EMC, but connected to the decimal carry input of the BPC. If DC is set, skip the amount indicated by n, or, to m.

label	SDC	* ± n/m	comments

Skip if Decimal Carry clear. Decimal Carry (DC) is a one-bit register in the EMC. It is controlled by the EMC, but connnected to the decimal carry input of the BPC. If DC is clear, skip the amount indicated by n, or, to m.

label	SHS	* ± n/m	comments

Skip if Halt line set. If the Halt line is true, skip the amount indicated by n, or, to m.

label	SHC	* ± n/m	comments

Skip if Halt line clear. If the Halt line if false, skip the amount indicated by n, or, to m.

label	SLA	* ± n/m [,S/,C]	comments

Skip if the least significant bit of A is zero. If the least significant bit (bit 0) of the A register is a zero, skip the amount indicated by n, or, to m. If either S or C is present, bit 0 is altered accordingly after the test.

ALTER-SKIP GROUP (CONT.)

label	SLB	* ± n/m [,S/,C]	comments

Skip if the least significant bit of B is zero. If the least significant bit (bit 0) of the B register is a zero, skip the amount indicated by n, or, to m. If either S of C is present, bit 0 is altered accordingly after the test.

label	RLA	* ± n/m [,S/,C]	comments

Skip if the least significant bit of A is non-zero. If the least significant bit (bit 0) of the A register is a one, skip the amount indicated by n, or, to m. If either S or C is present, bit 0 altered accordingly after the test.

label	RLB	* ± n/m [,S/,C]	comments

Skip if the least significant bit of B is non-zero. If the least significant bit (bit 0) of the B register is a one, skip the amount indicated by n, or, to m. If either S or C is present, bit 0 is altered accordingly after the test.

label	SAP	* ± n/m [,S/,C]	comments

Skip if A positive. If the sign bit (bit 15) of the A register is a zero, skip the amount indicated by n, or, to m. If either S or C is present, bit 15 is altered accordingly after the test.

label	SBP	* ± n/m [,S/,C]	comments

Skip if B positive. If the sign bit (bit 15) of the B register is a zero, skip the amount indicated by n, or, to m. If either S or C is present, bit 15 is altered accordingly after the test.

label	SAM	* ± n/m [,S/,C]	comments

Skip if A minus. If the sign bit (bit 15) of the A register is a one, skip the amount indicated by n, or, to m. If either S or C is present, bit 15 is altered accordingly after the test.

label	SBM	* ± n/m [,S/,C]	comments

Skip if B minus. If the sign bit (bit 15) of the B register is a one, skip the amount indicated by n, or, to m. If either S or C is present, bit 15 is altered accordingly after the test.

label	SOS	* ± n/m [,S/,C]	comments

Skip if overflow set. If the one-bit overflow register (OV) is set, skip the amount indicated by n, or, to m. If either S or C is present, the OV register is altered accordingly after the test.

ALTER-SKIP GROUP (CONT.)

label	SOC	* ± n/m [,S/,C]	comments

Skip if overflow clear. If the one-bit overflow register is clear, skip the amount indicated by n, or, to m. If either S or C is present, the OV register is altered accordingly after the test.

label	SES	* ± n/m [,S/,C]	comments

Skip if extend set. If the extend register (E) is set, skip the amount indicated by n, or, to m. If either S or C is present, E is altered accordingly after the test.

label	SEC	* ± n/m [,S/,C]	comments

Skip if extend clear. If the extend register (E) is clear, skip the amount indicated by n, or, to m. If either S or C is present, E is altered accordingly after the test.

COMPLEMENT-EXECUTE GROUP

label	CMA	comments

Complement A. The A register is replaced by its one's (bit by bit) complement.

label	CMB	comments

Complement B. The B register is replaced by its one's (bit by bit) complement.

label	TCA	comments

Two's complement A. The A register is replaced by its one's (bit by bit) complement, and then incremented by one. The E and OV registers are updated according to the results of the increment, in the same fashion as for the ADA instruction.

label	TCB	comments

Two's complement B. The B register is replaced by its one's (bit by bit) complement, and the incremented by one. The E and OV registers are updated according to the results of the increment, in the same fashion as for the ADB instruction.

COMPLEMENT-EXECUTE GROUP (CONT.)

label	EXE	$0 \leq m \leq 37_8$ [,I]	comments

Execute register m. The contents of any register can be treated as the current instruction, and executed in the normal manner. The register is left unchanged unless the instruction code causes it to be altered. The next instruction executed will be the one following the EXE m, unless the code in m causes a branch.

Indirect addressing is allowed. An EXE m, I causes the contents of m to be taken as the address of the place in memory whose contents are to be executed; this can be anywhere in memory, and need not be another register. In 15-bit versions of the processor, multi-level indirect addressing with EXE instruction is possible. Only one level is possible with the 16-bit processor.

The 15-bit verion of the BPC has a bug in connection with the Execute instruction. If the EXE machine-instruction is used to execute any of the A, B, P, or R registers, and interrupt occurs during the instruction fetch out of one of those registers, the BPC slips a cog and fails to give SMC (Synchronized Memory Complete). This failure to complete a memory cycle brings all system activity to a halt.

This bug is really not an exclusive property of the EXE instruction. The fundamental problem lies in *instruction fetches from addressable registers within the BPC*. An EXE instruction simply causes such a fetch. Such an unlikely thing as JMP A (although very legal and quite possible) would also suffer the uncompleted memory cycle if an interrupt were to occur during the fetch from A.

Note that EXE A ,I is not affected by the bug. Although it causes a read from A, that read is *not* an instruction fetch. It is only an instruction fetch from one of the addressable registers in the BPC that is susceptible to the bug. However, also note that an EXE A ,I is susceptible if A points to one of the other addressable registers with the BPC.

If the system uses interrupt it is best to disable the interrupt system with DIR before doing any EXE machine-instructions.

This bug has been fixed in the 16-bit version of the BPC.

STACK GROUP

The stack group manages first-in, last-out firmware stacks. The "place" instructions put a word or byte into a stack pointed at by C or D.* The item that is placed is reg. 0-7. The "withdraw" instructions remove a word or a byte from a stack pointed at by C or D. The removed item is written into reg. 0-7.

By the end of each place or withdraw instruction the stack pointer is either incremented or decremented, as specified by the optional I or D, respectively. In the absence of either an I or a D, the assembler defaults to I for place instructions, and D for withdraw instructions.

Place instructions increment or decrement the stack pointer prior to the placement, and withdraw instructions do it after the withdrawal. In this way the pointer is always left pointing at the top of the stack.

For byte operations using 15-bit version of the processor bit 15 of the pointer register (C or D) indicates left or right half (1 = left, 0 = right). Stack instructions involving bytes toggle bit 15 at each increment or decrement; but the lower bits of the pointer increment or decrement only during the zero-to-one transition of bit 15.

In the 16-bit version of the processor, the least-significant bit of the pointer register indicates left or right half (0 = left, 1 = right). Full 16-bit addressing is maintained by a most-significant bit (for each pointer register) in the form of the CB and DB registers. The C and CB registers, and D and DB registers, act as 17-bit registers during the automatic increment or decrement to the pointer registers.

The values of C and D for place-byte instructions must not be the address of any internal register for the BPC, EMC, or IOC. The place and withdraw instructions can also initiate I/O operations, so they are also listed under the I/O group.

label	PWC	reg. 0-7 [,I/,D]	comments

Place the entire word of reg. into the stack pointed at by C.

label	PWD	reg. 0-7 [,I/,D]	comments

Place the entire word of reg. into the stack pointed at by D.

label	PBC	reg. 0-7 [,I/,D]	comments

Place the right half of reg. into the stack pointed at by C.

label	PBD	reg. 0-7 [,I/,D]	comments

Place the right half of reg. into the stack pointed at by D.

* C and D are registers in the IOC; addresses 16_8 and 17_8, respectively.

IOC MACHINE INSTRUCTIONS

STACK GROUP (CONT.)

label	WWC	reg. 0-7 [,I/,D]	comments

Withdraw an entire word from the stack pointed at by C, and put it into reg.

label	WWD	reg. 0-7 [,I/,D]	comments

Withdraw an entire word from the stack pointed at by D, and put it into reg.

label	WBC	reg. 0-7 [,I/,D]	comments

Withdraw a byte from the stack pointed at by C, and put it into the right half of reg.

label	WBD	reg. 0-7 [,I/,D]	comments

Withdraw a byte from the stack pointed at by D, and put it into the right half of reg.

label	CBL*	comments

Set the CB register to a zero. This specifies the lower block of memory pointed at by C and CB.

label	CBU*	comments

Set the CB register to a one. This specifies the upper block of memory pointed at by C and CB.

label	DBL*	comments

Set the DB register to a zero. This specifies the lower block of memory pointed at by D and DB.

label	DBU*	comments

Set the DB register to a one. This specifies the upper block of memory pointed at by D and DB.

* Part of the 16-bit processor's instruction set only.

IOC MACHINE INSTRUCTIONS

I/O GROUP

The states of $\overline{IC1}$ and $\overline{IC2}$ during the I/O Bus Cycles initiated by the instructions below depend upon which register is the operand of the instruction:

	$\overline{IC1}$	$\overline{IC2}$
R4	1	1
R5	0	1
R6	1	0
R7	0	0

label	mem. ref. inst.	reg. 4-7 [,I]	comments

Initiate an I/O Bus Cycle. Memory reference instructions 'reading' from reg. cause input I/O Bus Cycles; those 'writing' to reg. cause output I/O Bus Cycles. In either case the exchange is between A or B and the interface addressed by the PA register (Peripheral Address Register - 11_8); reg. 4-7 do not really exist as physical registers within any chip on the IDA Bus.

label	stack inst.	reg. 4-7 [,I/,D]	comments

Initiate an I/O Bus Cycle. Place instructions 'read' from reg., therefore they cause input I/O Bus Cycles. Withdraw instructions 'write' into reg., therefore they cause output I/O Bus Cycles. In either case the exchange is between the addressed stack location and the interface addressed by PA.

INTERRUPT GROUP

label	EIR	comments

Enable the interrupt system. This instruction cancels DIR.

label	DIR	comments

Disable the interrupt system. This instruction cancels EIR.

DMA GROUP

label	SDO*	comments

Set DMA outwards. This instruction specifies the read-from-memory, write-to-peripheral direction for DMA transfers.

label	SDI*	comments

Set DMA inwards. This instruction specifies the read-from-peripheral, write-to-memory direction for DMA transfers.

label	DMA	comments

Enable the DMA mode. This instructions cancels PCM and DDR.

label	PCM	comments

Enable the Pulse Count Mode. This instruction cancels DMA and DDR.

label	DDR	comments

Disable Data Request. This instruction cancels the DMA Mode and the Pulse Count Mode.

> **NOTE**
>
> DDR is not usable with the 15-bit version of the processor. If the IOC should be in the process of executing a DDR and a DMA request occurs, the processor will go out to lunch and never come back. This bug has been fixed in the 16-bit version.

> **NOTE**
>
> The IOC will not execute IOC machine-instructions fetched from its own internal registers.

* Part of the 16-bit processor's instruction set only.

EMC MACHINE INSTRUCTIONS

THE FOUR-WORD GROUP

label	CLR	N	comments

Clear N words. This instruction clears N consecutive words, beginning with location < A >. Remember: $1 \le N \le 16_{10}$.

$$0 \rightarrow \text{location} < A >$$

$$0 \rightarrow \text{location} < A > + 1$$

$$\circ$$
$$\circ$$
$$\circ$$

$$0 \rightarrow \text{location} < A > + N - 1$$

label	XFR	N	comments

Transfer N words. This instruction transfers the N consecutive words beginning at location < A > to those beginning at < B >. Remember: $1 \le N \le 16_{10}$.

$$\text{location} < A > \rightarrow \text{location} < B >$$

$$\text{location} < A > + 1 \rightarrow \text{location} < B > + 1$$

$$\circ$$
$$\circ$$
$$\circ$$

$$\text{location} < A > + N - 1 \rightarrow \text{location} < B > + N - 1$$

THE MANTISSA SHIFT GROUP

label	MRX	comments

Mantissa right shift of ARI r-times, $r = < B_{0-3} >$, and $0 \le r \le 17_8 = 15_{10}$.

1st shift: $< A_{0-3} > \rightarrow D_1; \ldots .< D_i > \rightarrow D_{i+1}; \ldots .\ D_{12}$ is lost
jth shift: $0 \rightarrow D_1; \ldots .< D_i > \rightarrow D_{i+1}; \ldots .\ D_{12}$ is lost
rth shift: $0 \rightarrow D_1; \ldots .< D_i > \rightarrow D_{i+1}; \ldots .< D_{12} > \rightarrow A_{0-3};\ 0 \rightarrow DC;\ 0 \rightarrow A_{4-15}$

Notice:

1) The first shift does not necessarily shift in a zero; the first shift shifts in $< A_{0-3} >$.
2) The last digit shifted out ends up as $< A_{0-3} >$.
3) If only one digit-shift is done, (1) and (2) happen together.
4) After (2), SE is the same as $< A_{0-3} >$.
5) Any more than eleven shifts is wasteful.

THE MANTISSA SHIFT GROUP (CONT.)

label	MRY	comments

Mantissa right shift of AR2 $< B_{0-3} >$ -times. Otherwise identical to MRX.

label	MLY	comments

Mantissa left shift of AR2 one time.
$< A_{0-3} > \rightarrow D_{12};\ldots< D_i > \rightarrow D_{i-1};\ldots.< D_1 > \rightarrow A_{0-3};\ 0 \rightarrow DC;\ 0 \rightarrow A_{4-15}$

At the conclusion of the operation SE equals $< A_{0-3} >$.

label	DRS	comments

Mantissa right shift of ARI one time.
$0 \rightarrow D_1;\ldots.< D_i > \rightarrow D_{i+1};\ldots.< D_{12} > \rightarrow A_{0-3};\ 0 \rightarrow DC;\ 0 \rightarrow A_{4-15}$

At the conclusion of the operation SE equals $< A_{0-3} >$.

label	NRM	comments

Normalize AR2. The mantissa digits of AR2 are shifted left until $D_1 \neq 0$. If the original D_1 is non-zero, no shifts occur. If twelve shifts occur, then AR2 equals zero, and no further shifts are done. The number of shifts is stored as a binary number in B .

i. $0 \rightarrow B_{4-15}$; # of shifts $\rightarrow B_{0-3}$; $0 \rightarrow DC$
ii. For $0 \leq < B_{0-3} > \leq 11$; $0 \rightarrow DC$
iii. If $< B_{0-3} > = 12$; $1 \rightarrow DC$

THE ARITHMETIC GROUP

label	CMX	comments

Ten's complement of ARI. The mantissa of ARI is replaced with its ten's complement, and DC is set to zero.

NOTE

> In the 15-bit version of the processor there is a bug concerning CMX in 15-bit systems that also use DMA.

The bug concerns the way Sync is treated. Under the right conditions a bus grant (think "DMA cycle") causes the EMC to give Sync too early. The result is simultaneous use of the IDA Bus by the EMC and BPC. The most apparent result is that the next instruction fetch by the BPC is garbled, which is a disaster.

THE ARITHMETIC GROUP (CONT.)

label	CMY	comments

Ten's complement of AR2. The mantissa of AR2 is replaced with it ten's complement, and DC is set to zero.

label	CDC	comments

Clear Decimal Carry. Clears the DC register; 0 → DC.

label	FXA	comments

Fixed-point addition. The mantissas of AR1 and AR2 are added together, along with DC (as a D_{12} -digit), and the result is placed in AR2. If an overflow occurs, DC is set to one, otherwise, DC is set to zero at the completion of the addition.

During the addition the exponents are not considered, and are left strictly alone. The signs are also left completely alone.

< AR1 > = D_1 D_2 D_3--------D_{12}
< AR2 > = D_1 D_2 D_3--------D_{12}
+ < DC > ← initial value of DC

(overflow) → "D_0" D_1 D_2 D_3--------D_{12} → AR2
↳ DC (final value of DC)

label	MWA	comments

Mantissa Word Add. < B > is taken as four BCD digits, and added, as D_9 through D_{12}, to AR2. DC is also added in as a D_{12}. The result is left in AR2. If an overflow occurs, DC is set to one, otherwise, DC is set to zero at the completion of the addition.

During the addition the exponents are not considered, and are left strictly alone, as are the signs. MWA is intended primarily for use in rounding routines.

< B > = --------D_9 D_{10} D_{11} D_{12}
< AR2 > = D_1------D_9 D_{10} D_{11} D_{12}
+ < DC > ← initial value of DC

(overflow) → "D_0" D_1------D_9 D_{10} D_{11} D_{12} → AR2
↳ DC (final value of DC)

THE ARITHMETIC GROUP (CONT.)

label	FMP	comments

Fast multiply. The mantissas of AR1 and AR2 are added together (along with DC as D_{12}) $< B_{0-3} >$-times; the result accumulates in AR2.

The repeated additions are likely to cause some unknown number of overflows to occur. The number of overflows that occurs is returned in A_{0-3}.

FMP is used repeatedly to accumulate partial products during BCD multiplication. FMP operates strictly upon mantissa portions; signs and exponents are left strictly alone.

$$< AR2 > + ((< AR1 >) \cdot (< B_{0-3} >)) + DC \rightarrow AR2$$

DC doesn't enter into these repeated additions except for the first one as shown at right. 0 → DC immediately after each overflow.

Represents the initial value of DC.

$0 \rightarrow DC$, $0 \rightarrow A_{4-15}$ # of overflows $\rightarrow A_{0-3}$

label	MPY	comments

Binary Multiply Using Booth's Algorithm. The (binary) signed two's complement contents of the A and B registers are multiplied together. The thirty-two bit product is also a signed two's complement number, and is stored back into A and B. B receives the sign and most-significant bits, and A the least-significant bits:

$$< A > \cdot < B > \rightarrow \overbrace{< B > < A >}$$

NOTE

There is a bug in MPY. See the Appendix for its description.

label	FDV	comments

Fast Divide. The mantissas of AR1 and AR2 are added together until the first decimal overflow occurs. The result of these additions accumulates into AR2. The number of additions without overflow (n) is placed into B.

$$< AR2 > + < AR1 > + < DC > \rightarrow AR2 \quad \text{(repeatedly until overflow)}$$

then

$$0 \rightarrow DC, \quad 0 \rightarrow B_{4-15}, \quad n \rightarrow B_{0-3}$$

FDV is used in floating-point division to find the quotient digits of a division. In general, more than one application of FDV is needed to find each digit of the quotient.

As with the other BCD instructions, the signs and exponents of AR1 and AR2 are left strictly alone.

INTRODUCTION TO THE ASSEMBLER

GENERAL INFORMATION

The assembler (ASMA) translates symbolic source language instructions into an object program executable by the CPD processor. The source language provides mnemonic codes for specifying machine operations, (machine instructions) and for directing the assembler (pseudo instructions). The assembler also provides symbolic addressing. ASMA (July '76 version) serves both the 15 and 16 bit versions of the processor.

ASMA is a DOS-M or RTE based program; neither BCS nor MTS versions exist. DOS-M and RTE are disc operating systems for HP 2100-series computers. As of this writing there is also a series of 3000-based programs that assemble for the CPD processor. Presently several programs exist, each having different attributes. There is some sentiment to combine these programs. However, the move is not yet afoot, and the consensus was not to mention any program names or definite attributes. Generally speaking, the capabilities of the 3000-based assembler are much the same as those of ASMA, except that the DFN and $$$ pseudo instructions do not exist in the 3000 version. Also, the details of the "control statements" may differ. Generally, however, the two assemblers overlap about 95%; they are alike for more than they are different.

The assembled program is always "absolute" in the sense that it is not "relocatable"; the assembler assigns symbols definite addresses, and the operand fields of address-sensitive instructions receive definite bit patterns during assembly. If a piece of executable code is to be moved from one location to another, the usual case is that is must be modified to reflect the change in origin, and re-assembled. Assemblies must be self-contained: no external references (externals), entry points, or detached subroutines are possible.

With non-relocatability firmly in mind, we assign another meaning to the word absolute. The BPC has two modes of addressing: absolute and relative. Absolute addressing is a scheme with fixed page boundaries, and 1024 words per page. Relative addressing centers the page on the current value of the program location counter (P) in the BPC; the page boundaries change as P changes. The BPC operates in the absolute or relative addressing mode, depending upon the external grounding of a pin on the chip (RELA). It is expected that the two types of addressing will not be mixed. Complete descriptions of each addressing scheme are found in the chapter titled "DESCRIPTION OF THE PROCESSOR".

The assembler can assemble code for either absolute or relative addressing. This is controlled with the control statement at the beginning of the source text. See "ASSEMBLER INPUT AND OUTPUT", in this chapter.

The original source of a program will usually be paper tape or punched cards, although it is possible with DOS-M to create a source file on the disc directly from the system tele-printer. The assembler accepts paper tape, punched cards, magnetic tape, and disc source files as input. Magnetic tapes must be previosly generated by the operating system. Standard DOS-M provides disc source files, while source files are available with RTE systems that have a file manager.

INTRODUCTION TO THE ASSEMBLER

GENERAL INFORMATION (CONT.)

Assembler output is of two types: a listing and the non-relocatable binary. The listing can be generated on any "list device" in the system, but the binary should be punched on a punch device. ASMA does not have the ability to store the binary in the job-binary-area of the disc. Furthermore, it is un-advisable to write the binary to a standard tape transport with the idea of later use. DOS-M and RTE do not correctly handle non-relocatable binary, even when it is just "in transit".

A basic binary loader is required to load the binary output into the processor. The format of the binary output is shown in the section "ASSEMBLER INPUT AND OUTPUT"; the Appendix contains a discussion about binary loaders.

ASMA is a modification of ASMB; ASMB is the HP assembler for the 2100-series computers. Those who are familiar with the operation of ASMB under DOS-M or RTE will have no difficulty with ASMA. Some of the pseudo instructions of ASMB are missing from ASMA (those pertaining to relocatable assemblies), while some additional pseudo instructions have been added. See "PSEUDO INSTRUCTIONS".

A cross reference generator is available for use with ASMA. The name of this program is XRFA, and it runs with both DOS-M and RTE.

Additional information about the structure of the assembler is contained in the Appendix.

INSTRUCTION FORMAT

A source language statement consists of a label, an operation code, an operand, and comments. The label is used when needed as a reference by other statements. The operation code may be a mnemonic representing a machine-operation or an instruction to the assembler concerning the task of assembly itself. An operand may be an expression consisting of an alphanumeric symbol, a number, a special character, or any of these combined by arithmetic operations. Indicators may be appended to the operand to specify certain functions such as indirect addressing. The comments portion of the statement is optional.

STATEMENT CHARACTERISTICS

The field of the source statement appear in the following order:

Label   Opcode   Operand   Comments

One or more spaces separate the fields of a statement. An end-of-statement mark terminates the entire statement. On paper tape these marks are "return" and "line feed". A single space following the end-of-statement mark from the previous source statement is the null field indicator for the label field.

INTRODUCTION TO THE ASSEMBLER

INSTRUCTION FORMAT

STATEMENT CHARACTERISTICS (CONT.)

The characters that may appear in a statement are these:

A through Z

0 through 9

other valid label characters

. (period)

* (asterisk)

+ (plus)

- (minus)

, (comma)

(space)

Any other ASCII characters may appear in the Remarks field.

The letters A through Z, the numbers 0 through 9, the period, and certain other characters, may be used in an alphanumeric symbol. In the first position in the label field, an asterisk indicates a comment; in the operand field, it represents the value of the program location counter in arithmetic address expressions. The comma separates an expression and an indicator in the operand field.

Spaces separate fields of a statement. Within a field they may be used freely when following +, -, or , .

The maximum length of a statement varies, but is at most 80 characters. See "STATEMENT LENGTH" for a complete discussion.

LABEL FIELD

The label field indentifies the statement and may be used as a reference by other statements in the program. (That is, the label is a place holder for the address of a word that is used by other statements that concern, or operate on, that word.)

The field starts in position one of the statement; the first position following an end-of-statement mark for the preceding statement. It is terminated by a space. A space in position one is the null field indicator for the label field; the statement is unlabeled.

A label is symbolic. It may have one to five characters consisting of A through Z, 0 through 9, and the symbols shown on the next page. The first character must be non-numeric. A label of more than five characters could be entered on the source language tape, but the assembler flags this condition as an error and truncates the label to the left-most five characters.

INTRODUCTION TO THE ASSEMBLER

INSTRUCTION FORMAT

LABEL FIELD (CONT.)

A-Z	!	/	$
0-9	"	?	%
. (period)	#	@	&

Each label must be unique within the program; two or more statements may not have the same symbolic name.

Example:

Label	Operation	Operand	Comments
^†	LDA	-----	NO LABEL
.ABCD			VALID LABEL
.1234			VALID LABEL
A.123			VALID LABEL
.			VALID LABEL
1.AB			ILLEGAL LABEL - FIRST CHARACTER
			NUMERIC.
ABC123			ILLEGAL LABEL - TRUNCATED TO
			ABC12.
A*BC			ILLEGAL LABEL - ASTERISK NOT
			ALLOWED IN LABEL.
^ABC†			NO LABEL -THE ASSEMBLER ATTEMPTS
			TO INTERPRET ABC AS AN OPERATION
			CODE.

† The caret symbol, ^ , indicates the presense of a space.

An asterisk in position one indicates that the entire statement is a comment. Positions 2 through the end of the statement are available for use. See "STATEMENT LENGTH". An asterisk with the label field is illegal in any position other than one.

OPCODE FIELD

The operation code defines an operation to be performed by the processor or the assembler. The opcode field follows the label field and is separated from it by at least one space. If there is no label, the operation code may begin anywhere after position one. The opcode field is terminated by a space immediately following an operation code. Operation codes are organized in the following categories:

Machine Operation Codes

- BPC
  - Memory Reference
  - Shift-Rotate
  - Alter-Skip
  - Return-Complement-Execute

INSTRUCTION FORMAT

OPCODE FIELD (CONT.)

Machine Operation Codes (cont'd)

- IOC
  - I/O Control
  - Stack Operations
  - Interrupt
  - DMA
- EMC
  - Four-Word Operation
  - Mantissa-Shift
  - Arithmetic

Pseudo Operation Codes

- Assembler control
- Address and symbol definition
- Constant definition
- Storage allocation
- Assembly Listing Control

Machine operation codes are discussed in detail in the chapter titled "MACHINE INSTRUCTIONS".

OPERAND FIELD

The meaning and format of the operand field depend on the type of operation code used in the source statement. The field follows the opcode field and is separated from it by at least one space. It is terminated by a space except when the space follows <,>, <+>, <-> or, if there are no comments, by an end-of-statement mark.

The operand field may contain an expression consisting of one of the following:

- Single symbolic term
- Single numeric term
- Asterisk
- Combination of symbolic terms, and the asterisk joined by the arithmetic operators + and -.

An expression may sometimes be followed by a comma and an indicator.

The operands for certain instructions consists of a series of terms separated by commas.

INSTRUCTION FORMAT (CONT.)

SYMBOLIC TERMS

A symbolic term may be one to five characters consisting of A through Z, 0 through 9, or the other label characters. The first character must be non-numeric.

Example:

```
    LDA  A1234     VALID IF DEFINED
    ADA  B.1       VALID IF DEFINED
    JMP  ENTRY     VALID IF DEFINED
    STA  1ABC      ILLEGAL OPERAND FIRST CHARACTER
                   NUMERIC.
    STB  ABCDEF    ILLEGAL OPERAND MORE THAN FIVE
                   CHARACTERS.
```

Unless a symbol is pre-defined by the assembler, a symbol used in the operand field must be defined elsewhere in the program in one of the following ways:

As a label in the label field of a machine operation.

As a label in the label field of a BSS, ASC, DEC, OCT, DEF, ABS, EQU or REP pseudo operation.

The assembler assigns a value to a symbol when it appears in one of the above fields of a statement.

The symbols that are pre-defined by the assembler are shown in Table A-1. Information about modifying or adding to the list of pre-defined symbols is contained in the Appendix. With the exception of ARI, all these symbols refer to registers within the various elements of the system. The address of ARI depends upon whether the assembly is for a 15 or 16 bit processor.

The one bit registers, E (Extend) and OV (Binary Overflow), are located within the BPC. The one-bit register, DC (Decimal Carry - BCD overflow), is located within the EMC. These registers are not addressable; they are accessed through dedicated instructions. Therefore, their names are not pre-defined by ASMA.

A symbolic term may be preceded by a plus or minus sign. If preceded by a plus or no sign, the symbol refers to its associated value. If preceded by a minus sign, the symbol refers to the two's complement of its associated value. A single negative symbolic operand may be used only with the ABS pseudo operation.

INSTRUCTION FORMAT

SYMBOLIC TERMS (CONT.)

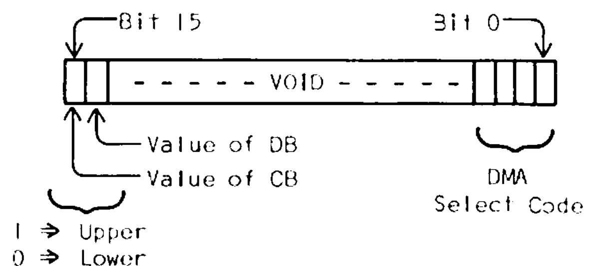
Table A-1. Symbols Pre-Defined by the Assembler.

Octal Address	Name	Location	Description (# of Bits)
0	A	BPC	Arithmetic Accumulator (16)
1	B	BPC	Arithmetic Accumulator (16)
2	P	BPC	Program Location Counter (least 15 of 16 or 16)
3	R	BPC	Return Stack Pointer (least 15 of 16 or 16)
4	R4	IOC	Peripheral Activity Designator (—)
5	R5	IOC	Peripheral Activity Designation (—)
6	R6	IOC	Peripheral Activity Designator (—)
7	R7	IOC	Peripheral Activity Designator (—)
10	IV	IOC	Interrupt Vector (upper 12 of 16)
* → 11	PA	IOC	Peripheral Address Register (least 4 of 16)
12	W	IOC	Working Register (16)
† → 13	DMAPA	IOC	2 MSB = CB & DB; 4 LSB = DMA Periph. Add. Reg.
14	DMAMA	IOC	DMA Memory Address & Direction Register (16)
15	DMAC	IOC	DMA Count Register (16)
16	C	IOC	Stack Pointer (16)
17	D	IOC	Stack Pointer (16)
20-23	AR2	EMC	BCD Arithmetic Accumulator (4 x 16)
24	SE	EMC	Shift Extend Register (least 4 of 16)
* → 25-27	X	EMC	Internal Arithmetic Register (3 X 16)
30-37	UNASSIGNED		
77770/ 177770	AR1	R/W	BCD Arithmetic Register (4 x 16)

* Not available for general use. Part of processes internal to a chip. It is best to pretend that these registers do not exist.

† Read register 13_8 produces:

CB and DB are actually discrete registers, and while they can only be read by reading R13, storinging into R13 will not alter their values. Use the CBL, CBU, DBL and DBU machine instructions for that purpose. CB and DB exist in the 16-bit version only.



INSTRUCTION FORMAT (CONT.)

NUMERIC TERMS

A numeric term may be decimal or octal. A decimal number is represented by one to five digits within the range ±32767. An octal number is represented by one to six octal digits followed by the letter B; (0 to 177777B).

If a numeric is preceded by a plus or no sign, the binary equivalent of the number is used in the object code. If preceded by a minus sign, the two's complement of the binary equivalent is used. A negative numeric operand may be used only with the RET, DEC, OCT, and ABS pseudo operations. The maximum value of a numeric operand depends on the type of machine or pseudo instruction.

THE ASTERISK

An asterisk in the operand field refers to the value in the program location counter at the time the source program statement is encountered.

EXPRESSIONS

The asterisk, symbols, and numbers may be joined by the arithmetic operators + and - to form arithmetic address expressions. The assembler evaluates an expression and produces a value in the object code.

Example:

```
Label   Operation  Operand      Comments
        LDA        SYM+6        ADD 6 TO THE VALUE OF SYM
        ADA        SYM-3        SUBTRACT 3 FROM THE VALUE OF SYM
         .
         .
         .
        JMP        *+5          ADD 5 TO THE CONTENTS OF THE
         .                      PROGRAM LOCATION COUNTER.
         .
         .
        STB        -A+C-4       ADD - VALUE OF A, THE VALUE OF C
         .                      AND SUBTRACT 4.
         .
         .
        STA        XTA-*        SUBTRACT VALUE OF PROGRAM
                                LOCATION COUNTER FROM VALUE OF
                                XTA.
```

INSTRUCTION FORMAT

EXPRESSIONS (CONT.)

An expression consisting of a single term has the value of that term. An expression consisting of more than one term is reduced to a single value. In expressions containing more than one operator, evaluation of the expression proceeds from left to right. The algebraic expression A-(B-C+5) must be represented in the operand field as A-B+C-5. Parentheses are not permitted in expressions for the grouping of terms.

The range of values tolerated by the assembler during the evaluation of an expression depends upon the type of operation, and whether the assembly is for a 15-bit or 16-bit processor.

INDIRECT ADDRESSING

The processor provides an indirect addressing capability for memory reference instructions. The operand portion of an indirect instruction contains an address of another location rather than an actual operand. For 15-bit processors the secondary location may be the actual operand or it may be indirect also, and give yet another location, and so forth. The chaining ceases when a location is encountered that does not contain an indirect address.* Only the initial indirect reference is possible with 16-bit processors; the first address accessed indirectly contains a 16-bit destination address. Indirect addressing provides a simplified method of address modification as well as allowing access to any location in memory.

The assembler allows specification of indirect addressing by appending a comma and the letter I to any memory reference operand. The actual operand of the instruction may be given in a DEF pseudo operation; this pseudo operation may also be used to indicate further levels of indirect addressing (for 15-bit processors).

BASE PAGE AND CURRENT PAGE ADDRESSING

The processor provides a capability which allows the memory reference instructions to address either the "current page" or the "base page". The assembler adjusts all instructions in which the operands refer to the base page; specific notation defining an operand as a base page reference is not required in the source program. Any memory reference instruction; regardless of where in memory it is stored, can reference an address on the base page. Things not located on the base page are located on one of many different current pages. A direct reference to a location not on the base page is possible only if the instruction making the reference is on the same (current) page as the referenced location.

COMMENT FIELD

The comment field allows the programmer to transcribe notes that will be included with the source language coding on the list output produced by the assembler. The comment field follows the operand field, and is separated from it by at least one space.

* For 15-bit processors such an indirect address in memory is indicated by a one in bit 15; bits 0-14 contain the address that is indirect. A non-indirect address has a zero in bit 15.

INSTRUCTION FORMAT

COMMENT FIELD (CONT.)

The comment field is terminated by the end-of-statement mark, or by indirect means within DOS-M or the assembler itself. See the discussion in the next section.

On listing, statements consisting entirely of comments begin in position 27. Other statements begin in position 21. (The numbering assumes the first position is named 1.) This shifts the comment to the right so that the label field column in the listing produced by the assembler is free of anything except labels and errors. This makes it easier to look for and find a label in the listing.

STATEMENT LENGTH

The maximum length of a statement that is not a comment is 80 characters. Comment statements are limited to 74 characters.

Punched cards limit the length of a statement to what can be put on a single card; there is no continuation-card mechanism. This limits a statement to 80 characters, the end of the card acts as an end-of-statement mark.

If the source was originally paper tape which was then stored as a source file on DOS-M, it was truncated to a maximum of 80 characters per line by DOS-M at that time. RTE has no such truncation mechanism, but the assembler still limits the length of a statement to 80 characters.

The assembler can read the source text directly from paper tape; the same restrictions on length apply.

Characters beyond the limits are ignored, and not printed on the listing.

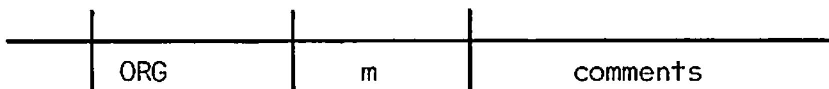
ASSEMBLER PSEUDO INSTRUCTIONS

The pseudo instructions control the assembler, as well as specify various types of constants, blocks of memory, and labels used in the program. Pseudo instructions also control the listing.

ASSEMBLER CONTROL

The assembler control pseudo instructions establish and alter the contents of the program location counter, and terminate assembly processing. Labels may be used but they are ignored by the assembler.

ORG AND ORR



The ORG statement defines the origin (initial value of the program counter) of a program, or the origins of subsequent sections of programming.

Generally, a program begins with an ORG statement.* An ORG statement must preceed any machine instructions. The operand, m, must be a decimal or octal integer specifying the initial setting of the program location counter.

ORG statements may be used elsewhere in the program to define starting addresses for portions of the object code; the operand field, m, may be any expression. Symbols in the operand must be previously defined. All instructions following an ORG are assembled at consecutive addresses starting with the value of the operand. For 15-bit assemblies the maximum value of the operand is 77777B. The value of the operand is not restrained for 16-bit assemblies.

ORR is an automatic reset of the value of the assembler's program location counter. Its action is described below.

The assembler traps the very first value given to the program location counter (by the first ORG in the program). Thereafter, as the value of the program location counter is incremented from that initial value by "natural consumption" of address space (any in-line code except ORG's), a duplicate copy of the current value of the program location counter is maintained. An ORG subsequent to the first one causes the duplicate value to be saved, and the updating mechanism to be turned off.

* The Control Statement, the HED instruction, and comments may appear prior to the ORG statement. See "ASSEMBLER INPUT AND OUTPUT" for a description of the Control Statement.

ASSEMBLER CONTROL

ORG AND ORR (CONT.)

An ORR causes the program location counter to be re-set to its earlier value (that of the duplicate), and also re-invokes the mechanism for maintaining the duplicate, so that the process can be repeated for other ORG -- ORR pairs.

Example:

```
0001  ASMB,A,L,C
0002        HED ORR TEST
0003        ORG 100B      INITIAL VALUE OF PLC
0004        NOP
0005        NOP
0006        ORR           NO EFFECT, NO SECOND ORIGIN
0007        NOP
0008        NOP
0009        SPC 1
0010        ORG 200B      SECOND OR LATER ORIGIN
0011        NOP
0012        NOP
0013        SPC 1
0014        ORG 300B
0015        NOP
0016        NOP
0017        SPC 1
0018        ORR           RESET ORIGIN
0019        NOP
0020        NOP
0021        ORR           NO EFFECT ON PLC
0022        NOP
0023        NOP
0024        SPC 1
0025        ORG 400B
0026        NOP
0027        NOP
0028        SPC 1
0029        ORR           RESET ORIGIN AGAIN
0030        NOP
0031        NOP
0032        END
**** LIST END ****
```

NEW INSTRUCTION DEFINITION

ASMA allows the user to define, at assembly time, his own custom machine instructions. The definitions must precede the use of such custom instructions, and are in force for the duration of that assembly only. ASMA allows up to 70 custom instructions to be defined at one time.

ASSEMBLER CONTROL

NEW INSTRUCTION DEFINITION (CONT.)

DFN	mnemonic,	type,	bit pattern	comments

Defines a machine instruction with the given 3-character mnemonic, with the given basic bit pattern, and whose general properties (in terms of its assembler-generated bit fields) is one of the types shown in the bit pattern tabulations in the Appendix.

During the assembly of a program, an instruction in the source coding is identified by matching it against a table in the assembler. The permanent instruction table is searched first, followed, if necessary, by a search of table space generated by DFN's. Because of the order of this search, DFN cannot be used to re-define existing instructions.

Each of the fields in the source DFN instruction may be preceded by leading blanks on the left. Trailing blanks between the substance of the field and the indicated comma are not permitted.

The type and bit pattern fields are assumed to represent octal integers; do not follow them with a B.

Only existing "types" may be used in DFN instructions; see the tabulation of types and bit patterns in the Appendix. There is no protection against using an undefined or inappropriate type. To do so, however, is a sure-fire way to send the assembler out-to-lunch.

Each generic type of manipulation performed by the assembler, as it produces an instruction, is represented by a number called the "type". The type field tells the assembler how to handle the newly defined instruction. All instructions of a given type are processed identically, except for their differences in their basic bit patterns. New types cannot be defined without modifying the source of the assembler itself.

The following two examples illustrate the properties of "type".

For instance, type 30 instructions never have operands or modifiers like ,I. Such an instruction has a fixed 16-bit pattern, and every occurrence of that instruction results in exactly that particular pattern. The majority of the Math Chip instructions, and some of the I/O Chip instructions are type 30 instructions. Type 30 instructions work in either 15-bit or 16-bit assemblies. Type 46 instructions are identical to type 30 instructions, except that they are allowed only in 16-bit assemblies.

Memory reference instructions are type 16, and are perhaps the most complicated type of instruction. The action of a type 16 instructions is as follows:

ASSEMBLER CONTROL

NEW INSTRUCTION DEFINITION (CONT.)

1) Evaluates an operand, allowing expressions and a ,I.
2) Checks the resulting value for admissability, based on its value vis-à-vis the program location counter (is the operand on the current page, base page, or neither?).
3) Sets the B/C bit (bit 10) according to whether or not the operand is on the base page.
4) Creates a certain type of 10-bit reference to the operand and "or's" it (in the bit 0-9 positions) with the basic bit pattern.
5) Checks for a ,I following the operand, and sets bit 15 of the instruction if there was one.

It is a characteristic of the assembler that it "or's" the value of any computed operand into the supplied basic bit pattern. If an instruction is to have a four-bit field in bits 0-3, the basic bit pattern must be zeros in those bits. Likewise, any bit that is to be set by a comma I, or other modifier, must also be a zero in the basic bit pattern.

Now, type 16 is closed, and not available for use if the processor includes a BPC (a most likely state of affairs). This is because this type allows only bits 11-14 as basic bit pattern, and 14 of the 16 possible combinations specify existing memory reference instructions in the BPC. The other combinations are necessary ingredients of any non-memory-reference instruction.

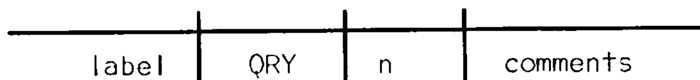
Examples:

DFN QRX, 30, 076543

This defines an instruction whose name is QRX and whose basic bit pattern is 076543 octal, with no operands or modifiers allowed.

DFN QRY, 27, 076560

This defines an instruction whose name is QRY and whose bit pattern is 076560 merged with a 4-bit field in bits 0-3. Other than for the basic bit pattern, QRY is the same as a shift-rotate instruction, as far as ASMA is concerned. QRY would be described thusly:



QRY sets the brass-plated knudsen valve to one of 16 positions, depending upon the value of n; n may range from 1 to 16 in source, bits 0-3 are encoded with the binary for n-1.

Good Luck!

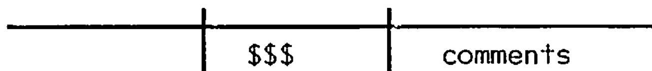
PARTITIONING A BINARY TAPE

The assembler provides the capability to arbitrarily insert long sections of feed-frames in the output binary tape. This causes the loader to stop. By

ASSEMBLER CONTROL

PARTITIONING A BINARY TAPE (CONT.)

utilizing this feature, several sections of independent code can be assembled together, but loaded separately, or in a different order.

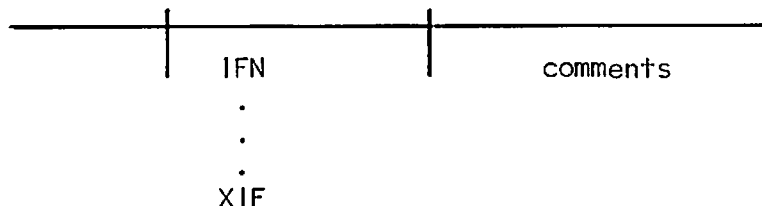


Causes any binary generated to this point to be properly outputted as a complete record. Then causes the punching of 90 feed-frames (9 inches). Such a break causes the binary leader to stop loading at that point. It also allows easy visual identification of the sections of a binary tape.

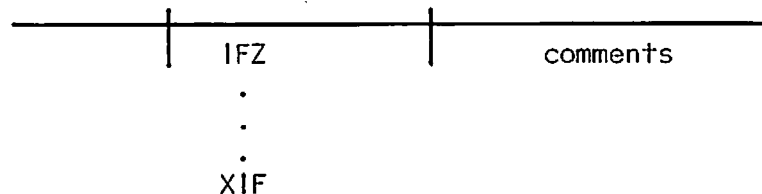
$$$ may be used anywhere in a program without disturbing the validity of the resulting binary records on either side of the inserted feed-frames.

CONDITIONAL ASSEMBLY

The IFN and IFZ pseudo instructions cause the inclusion of instructions in a program provided that either an "N" or "Z", respectively, is specified as a parameter in the control statement. The IFN or IFZ instruction precedes the set of statements that are to be included. The pseudo instruction XIF serves as a terminator. If XIF is omitted, END acts as a terminator to both the set of statements and the assembly.



All source language statements appearing between the IFN and the XIF pseudo instructions are included in the program if the character "N" is specified in the ASMB control statement.



All source language statements appearing between the IFZ and the XIF pseudo instructions are included in the program if the character "Z" is specified in the ASMB control statement.

When the particular letter is not included on the control statement, the related set of statements appears on the assembler output listing but is not assembled.

ASSEMBLER CONTROL

CONDITIONAL ASSEMBLY (CONT.)

Any number of IFN-XIF and IFZ-XIF sets may appear in a program; however, they may not overlap. An IFZ or IFN intervening between an IFZ or IFN and the XIF terminator results in a diagnostic being issued during assembly; the second pseudo instruction is ignored.

Both IFN-XIF and IFZ-XIF pseudo instructions may be used in the program; however, only one type will be selected in a single assembly. If both characters "N" and "Z" appear in the control statement, the character which is listed last will determine the set of coding that is to be included in the program.

Examples:

```
0001  ASMB,A,L,N,B
0002          .
0003          .
0004          .
0005        IFN
0006        DFN QRX, 30, 123456      DEFINE QRX
0007          .
0008          .
0009          .
0010  HOOK  QRX
0011        JMP ZAPIT
0012        XIF
0013          .
0014          .
0015          .
0016        END
**** LIST END ****
```

```
0001  ASMB,A,L,B,Z
0002          .
0003          .
0004          .
0005        IFZ
0006  * NOW BREAK BINARY TAPE IF Z IN CNTRL STMT
0007        SSS
0008        XIF
0009          .
0010          .
0011          .
0012        END
**** LIST END ****
```

ASSEMBLER CONTROL (CONT.)

AUTOMATIC STATEMENT REPETITION

The REP pseudo instruction causes the repetition of the statement immediately following it a specified number of times.

The statement following the REP in the source program is repeated n times. The n may be any expression. Comment lines (indicated by an asterisk in character position 1) are not repeated by REP. If a comment follows a REP instruction, the comment is ignored and the instruction following the comment is repeated.

A label specified in the REP pseudo instruction is assigned to the first repetition of the statement. A label cannot be part of the instruction to be repeated; it would result in a doubly defined symbol error.

Example:

```
         CLA
TRIPL    REP      3
         ADA      DATA
```

The above source code would generate the following:

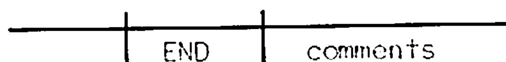
```
         CLA               Clear the A-Register;
TRIPL    ADA      DATA     the contents of DATA
         ADA      DATA     is tripled and stored in
         ADA      DATA     the A-Register.
```

Example:

```
FILL     REP      100B
         NOP
```

The example above loads 100_8 memory locations with the NOP instruction. The first location is labeled FILL.

SOURCE TERMINATION



This statement terminates the program; it marks the physical end of the source language statements.

The label field of the END statement is ignored.

ADDRESS AND SYMBOL DEFINITION

The pseudo operations is this group assign a value or a word location to symbol which is used as an operand elsewhere in the program.

label	DEF	m [,I]	comments

The address definition statement generates one word of memory as a 15-bit or 16-bit address which may be used as the object of an indirect address found elsewhere in the source program. The symbol appearing in the label is that which is referenced; it appears in the operand field of a memory reference instruction.

The operand field of the DEF statement may be any positive expression.

The expression in the operand field may itself be indirect and make reference to another DEF statement elsewhere in the source program. The ,I causes the assembler to set the 16th bit of the generated word. This feature is not illegal in 16-bit assemblies, although it really only makes sense to do it in 15-bit assemblies.

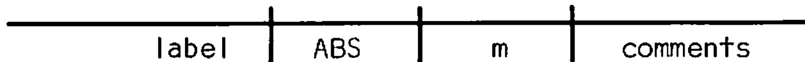
Examples:

```
0001          LDA  INST     A IS LOADED WITH ADDRESS OF BUFFR+3
0002               .
0003               .
0004               .
0005   LABEL  DEF  BUFFR
0006   INST   DEF  BUFFR+3
0007               .
0008               .
0009               .
0010          ORG  77000B
0011   BUFFR  BSS  40
0012               .
0013               .
0014               .
**** LIST END ****
```

ADDRESS AND SYMBOL DEFINITION (CONT.)

Example (cont'd)

```
0001        LDA HOOK,I    A GETS LOADED WITH 171717
0002          .
0003          .
0004          .
0005  HOOK  DEF ROOK,I    THE ,I SETS BIT 15 OF HOOK
0006          .
0007          .
0008          .
0009  ROOK  DEF ZIPPR
0010          .
0011          .
0012          .
0013  ZIPPR OCT 171717
0014          .
0015          .
0016          .
```

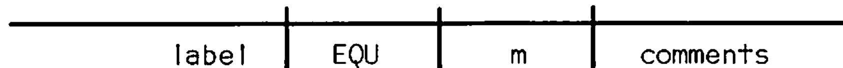


ABS defines a 16-bit value to be stored at the location represented by the label. The operand field, m, may be any expression or single symbol.

Example:

```
AB    EQU  35       ASSIGNS THE VALUE OF 35
                    TO THE SYMBOL AB

M35   ABS  -AB      M35 CONTAINS -35.
P35   ABS  AB       P35 CONTAINS 35.
P70   ABS  AB+AB    P70 CONTAINS 70.
P30   ABS  AB-5     P30 CONTAINS 30.
```



The EQU pseudo operation assigns to a symbol a value other than the one normally assigned by the program value represented by the operand field. The operand field may contain any expression. The value of the operand may not be negative. Symbols appearing in the operand must be previously defined in the source program.

The EQU instruction may be used to symbolically equate two locations in memory; or it may used to give a value to a symbol. The EQU statement does not result in a machine instruction.

ADDRESS AND SYMBOL DEFINITION (CONT.)

Example:

```
       .
       .
       .
J3     DEF
       .
       .
       .
       LDA  J3        THE SYMBOLS JFOUR AND J3+1 BOTH
       ADA  ONE       IDENTIFY THE SAME LOCATION. THE
       STA  J3+1      AND OPERATION IS PERFORMED ON
JFOUR  EQU  J3+1      THIS LOCATION.
       .
       .
       .
MWH    AND  JFOUR
       .
```

CONSTANT DEFINITION

The pseudo instructions in this class enter a string of one or more constant values into consecutive words of the object program. The statements may be named by labels; this allows other program statements to refer to the strings of words generated by them.

label	ASC	n, <2n characters>	comments

ASC converts a string of 2n alphanumeric characters in ASCII code into n consecutive words.* One character is right justified in each eight bits; the most significant bit is zero. n may be any expression resulting in an unsigned decimal value in the range 1 through 28. Symbols used in an expression must be previously defined. Anything in the operand field following 2n characters is treated as comments. If less than 2n characters are detected before the end-of-statement mark, the remaining characters are assumed to be spaces, and are stored as such. The label represents the address of the first two characters.

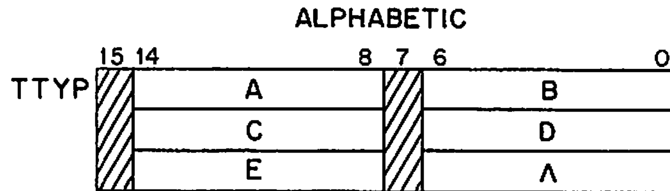
Example:

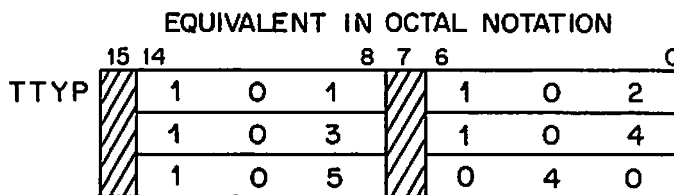
```
TTYP   ASC  3,ABCDE
```

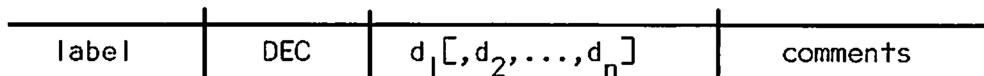
* To enter the code for the ASCII symbols which perform some action (e.g., "carriage return" and "line feed"), the OCT pseudo instruction must be used.

CONSTANT DEFINITION (CONT.)

causes the following:

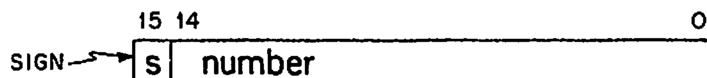




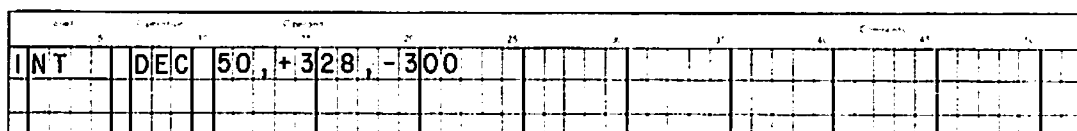


DEC records a string of decimal constants into consecutive words. The constants must be integers. If no sign is specified, positive is assumed. The decimal number is converted to its binary equivalent by the assembler. The label, if given, serves as the address of the first word occupied by the constant.

The decimal integer must fall within the following range: -32768 to 32767, including zero. Absolute values of 32769 or greater result in an error. Avoid ±32768. It results in the same binary result as for -32768; namely, 100000. Each decimal integer appears as one binary word and appears as follows:



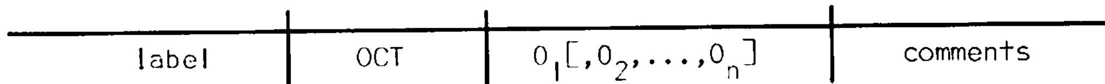
Example:



CONSTANT DEFINITION (CONT.)

causes the following (octal representation):

	15	14				0
INT	0	0	0	0	6	2
	0	0	0	5	1	0
	1	7	7	3	2	4



label	OCT	$0_1[,0_2,\ldots,0_n]$	comments

OCT stores one or more octal constants in consecutive words of the object program. Each constant consists of one to six octal digits (0 to 177777). If no sign is given, the sign is assumed to be positive. If the sign is negative, the two's complement of the binary equivalent is stored. The constants are separated by commas; the last constant is terminated by a space. If less than six digits are indicated for a constant, the data is right justified in the word. A label, if used, acts as the address of the first constant in the string. The letter B must not be used after the constant in the operand field.

Example:

Label	Opcode	Operand	Comments
	OCT	+Ø	
	OCT	-2	
NUM	OCT	177,2Ø4Ø5,-36	
	OCT	51,77777,-1,1Ø1Ø1	
	OCT	1Ø7642,177Ø77	
	OCT	1976	ILLEGAL: CONTAINS DIGIT 9
	OCT	-177777	
	OCT	177B	ILLEGAL: CONTAINS CHARACTER B

CONSTANT DEFINITION (CONT.)

The previous statements are stored as follows:

	15	14				0
	0	0	0	0	0	0
	1	7	7	7	7	6
NUM	0	0	0	1	7	7
	0	2	0	4	0	5
	1	7	7	7	4	2
	0	0	0	0	5	1
	0	7	7	7	7	7
	1	7	7	7	7	7
	0	1	0	1	0	1
	1	0	7	6	4	2
	1	7	7	0	7	7
	X	X	X	X	X	X
	0	0	0	0	0	1
	X	X	X	X	X	X

THE RESULT OF ATTEMPTING TO DEFINE AN ILLEGAL CONSTANT IS UNPREDICTABLE

STORAGE ALLOCATION

The storage allocation statement reserves a block of memory for data or for a work area.

label	BSS	m	comments

The BSS pseudo operation advances the program location counter according to the value of the operand. The operand field may contain any expression that results in a positive integer. Symbols, if used, must be previously defined in the program. The label, if given, is the name assigned to the storage area and represents the address of the first word. The initial content of the area set aside by the statement is unaltered by the loader.

ASSEMBLY LISTING CONTROL

Assembly listing control pseudo instructions allow the user to control the assembly listing output during the assembly process.

	UNL	comments

Output is suppressed from the assembly listing, beginning with the UNL

ASSEMBLY LISTING CONTROL (CONT.)

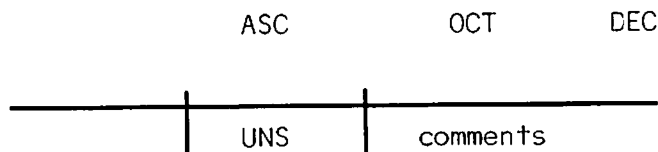
pseudo instruction and continuing for all instructions and comments until either an LST or END pseudo instruction is encountered. Diagnostic messages for errors encountered by the assembler will be printed, however. The source statement sequence numbers (printed in columns 1-4 of the source program listing) are incremented for the instructions skipped.

The LST pseudo instruction causes the source program listing, terminated by a UNL, to be resumed.

A UNL following a UNL, a LST following a LST, and a LST not preceded by a UNL are not considered errors by the assembler.

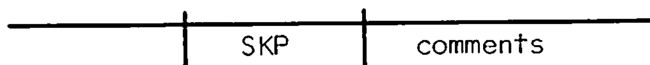
The SUP pseudo instruction suppresses the output of additional code lines from the source program listing. Certain pseudo instructions generate more than one line in the listing. These additional lines are suppressed by a SUP instruction until a UNS or the END pseudo instruction in encountered. SUP will suppress additional lines in the following pseudo instructions:

ASC OCT DEC

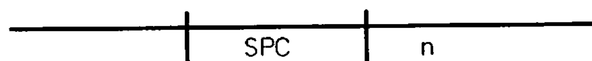


The UNS pseudo instruction causes the printing of additional listing lines, terminated by a SUP, to be resumed.

A SUP preceded by another SUP, UNS preceded by UNS, or UNS not preceded by a SUP are not considered errors by the assembler.

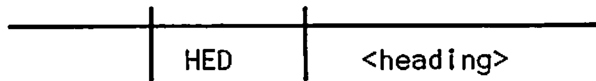


The SKP pseudo instruction causes the source program listing to skip to the top of the next page. The SKP instruction is not listed, but the source statement sequence number is incremented for the SKP.



The SPC pseudo instruction causes the source program listing to include a specified number of blank lines. The list output skips n blank lines, or to the bottom of the page, whichever occurs first. The n may be any absolute expression. The SPC instruction itself is not listed, but the source statement sequence number is incremented.

ASSEMBLY LISTING CONTROL (CONT.)



The HED pseudo instruction allows the programmer to specify a heading to be printed at the top of each page of the source program listing.

The heading, m, (a string of up to 56 ASCII characters), is printed at the top of each page of the source program listing following the occurrence of the HED pseudo instruction. If HED is encountered before the ORG at the beginning of a program, the heading will be used on the first page of the source program listing. A HED instruction placed elsewhere in the program causes a skip to the top of the next page.

The heading specified in the HED pseudo instruction will be used on every page until it is changed by a succeeding instruction.

The source statement containing the HED will not be listed, but source statement sequence number will be incremented.

ASSEMBLER INPUT AND OUTPUT

The assembler accepts as input: paper tape; punched cards; magnetic tape; disc source files. The output produced by the assembler consists of a listing containing diagnostics, and a punched paper tape containing the object program. The assembler can also automatically begin the execution of the cross reference program, following the assembly.

THE CONTROL STATEMENT

The control statement specifies whether to assemble for 15-bit or 16-bit processors, and specifies the output to be produced by the assembler.

ASMB,P_1,P_2,---,P_n

"ASMB," is entered in positions 1 through 5. Following the comma are one or more parameters, in any order, which define the output to be produced. The parameters may be any legal combination of the following, starting in position 6:

F — Fifteen-bit: The assembler assembles for processors that utilize 15-bit addressing.

S — Sixteen-bit: The assembler assembles for processors that utilize 16-bit addressing.

A — Absolute: The assembler assembles for fixed-page addressing; the 10-bit address fields for memory reference instructions are generated according to the absolute addressing scheme.

R — Relative: The assembler assembles for relative-page addressing; the 10-bit address fields for memory reference instructions are generated according to the relative addressing scheme.

B — Binary Output: The non-relocatable object program (which is either absolute or relative) is punched on the punch device.

L — Program Listing: A program listing is produced on the list device. The listing is annotated with diagnostics, should errors be detected in the program during assembly.

T — Symbol Table Listing: A listing of the symbol table generated by the assembler is produced. This listing precedes a program listing, regardless of the order of the respective parameters. The symbol table listing occurs in the order the symbols are defined, beginning with pre-defined symbols.
Do not confuse this listing with the cross reference. This listing is produced by the assembler; the cross reference is produced by a separate program, callable by the assembler, and also as a stand alone program by the user.

N — Include sets of instructions following the IFN pseudo instruction.

Z — Include sets of instructions following the IFZ pseudo instruction.

C — Begin the cross reference program (XRFA) immediately after assembly.

THE CONTROL STATEMENT (CONT.)

Either F or S must be specified. Likewise either A or R must be specified. Also, one of B, L, or T must be specified. Specifying C is optional. Also the control statement must be the very first statement in the program.

THE SOURCE PROGRAM

The first statement of a program must be a control statement; no other control statements are allowed in the program. The next statement required before assembly can proceed in an ORG statement. However, HED and comment statements can occur between the control statement and the first ORG statement. But no other types of statements may precede the first ORG. The last statement must be an END statement.

THE LISTING

Fields of the object program are listed in the following print columns.

Columns	Content
1	Blank
2-5	Source statement sequence number generated by the assembler
6	Blank
7-12	Location (octal)
13	Blank
14-19	Object code word in octal
20	Blank
21-100	First 80 characters of source statement

Lines consisting entirely of comment (i.e., *in column 1) are printed as follows:

Columns	Content
1	Blank
2-5	Source statement sequence number
27-100	Up to 74 characters of comment

A symbol table listing has the following format:

Columns	Content
1	Blank
2-6	Symbol
7-8	Blank
9-14	Value of the symbol

THE LISTING (CONT.)

Internally, ASMA is a two-pass process. During the first pass a symbol table is generated, and if the source is from a device other than the disc, the source is read onto the work area of the disc, in preparation for the second pass. It is at the end of the first pass that a listing of the symbol table is printed, if requested. The second pass generates the program listing and the actual object program (binary tape).

At the end of each pass, the following is printed:*

```
**NO ERRORS*
     or
**nnnnERRORS*
```

The value nnnn indicates the number of errors.

BINARY OUTPUT

A binary output tape consists of a series of records; each record has the format shown below. Records vary in length, but are maximum of 67_{10} words long.

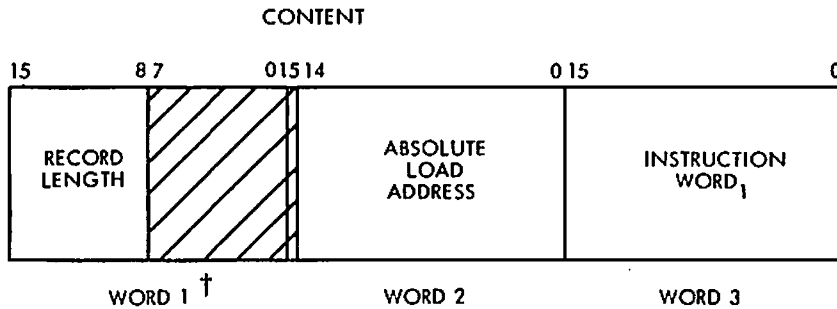
During the second pass of assembly, the object binary is accumulated in a buffer. The contents of the buffer will become a record on the output tape. A record is punched when the buffer gets full, or when it is necessary to begin a new record. Instructions like ORG, BSS and $$$ always cause the accumulated previous record to be punched (unless the buffer was empty), and a new record started.

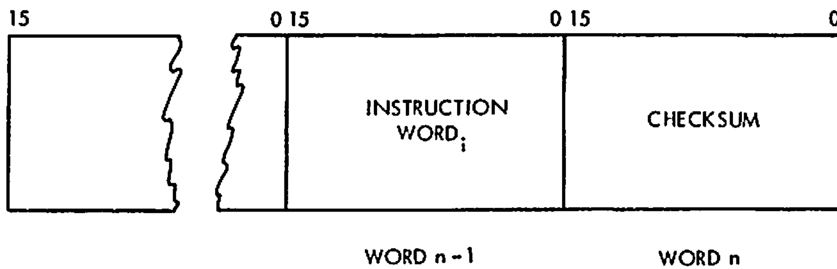
* The numbers refer to the number of errors detected during each pass only; it is possible for either number to be zero while the other is not. Always check both numbers, not just the one at the end of the listing. Also, pass one error diagnostics are simply printed, by themselves, at the start of the listing; they include the error mnemonic as well as the offending statement. Pass two error diagnostics are merged with the listing proper; the diagnostic itself has the same form as for pass one, but immediately preceeds regular listing of the offending statement. It is possible for a defective statement to produce more than one diagnostic message.

BINARY OUTPUT (CONT.)

OBJECT TAPE FORMAT

CONTENT





EXPLANATION

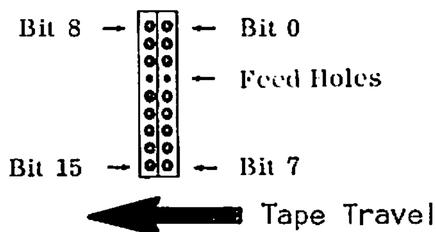
RECORD LENGTH = NUMBER OF WORDS IN RECORD EXCLUDING WORDS 1 AND 2 AND THE LAST WORD.

ABSOLUTE LOAD ADDRESS: STARTING ADDRESS FOR LOADING THE INSTRUCTIONS WHICH FOLLOW

INSTRUCTION WORDS: ABSOLUTE INSTRUCTIONS OR DATA

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS EXCEPT FIRST AND LAST

†Each word represents two frames arranged as follows:



APPENDIX

ASSEMBLER ERROR MESSAGES

During the assembly of a program, error messages are printed on the list output device to aid the programmer in debugging programs. Errors detected in the source program are indicated by 1- or 2- letter mnemonic followed by the sequence number and the first 62 characters of the statement in error. The messages are printed on the output device during the passes indicated.

Error Code	Pass	Description
CS	1	Control statement error: a) The control statement contains a parameter other than one in the legal set. b) Neither A nor R, or both A and R, are specified. c) Neither S nor F, or both S and F, are specified. d) There is no output parameter (B, T, or L).
DD	1	Doubly defined symbol: A name defined in the symbol table appears more than once as: a) A label of a machine instruction. b) A label of one of the pseudo operations: BSS EQU ASC ABS DEC OCT DEF
FU	1	Too many DFN statements.
IF	1	An IFZ or an IFN follows either an IFZ or an IFN without an intervening XIF. The second pseudo instruction is ignored.
IL	2	Illegal character: A numeric term used in the operand field contains an illegal character (e.g. an octal constant contains other than +,-, or 0-7).
M	1,2	Illegal operand: a) Operand is missing for an opcode requiring one. b) A negative operand is used with an opcode field other than ABS, or OCT. c) A character other than I follows a comma in one of the following statements: LDA ADB AND LDB STA DSZ CPA STB IOR CPB JSM JMP ADA ISZ DEF d) A character other than S or C follows a comma in one of the following statements: SLA SAM SLB SBM RLA SOS RLB SOC SAP SES SBP SEC e) An illegal operator appears in an operand field (e.g. + or - as the last character). f) An integer expression in an instruction does not meet a size requirement.
NO	1,2	No origin definition: The first statement in the assembly containing a valid opcode following the ASMB Control Statement (and remarks and/or HED, if present) is not an ORG statement.

ASSEMBLER ERROR MESSAGES (CONT.)

Error Code	and Pass	Description
OP	1,2	Illegal opcode preceding first valid opcode. Also, a comment fails to not contain an asterisk in position one. The statement is assumed to contain an illegal opcode; it is treated as a remarks statement.
OP	1,2	Illegal opcode: A mnemonic appears in the opcode field which is not valid. A word is generated in the object program, however.
OP	2	Opcode is valid in 16-bit assemblies, but invalid in present 15-bit assembly.
OV	1,2	Numeric operand overflow. The numeric value of a term or expression has overflowed its limit.
SO	1	There are more symbols defined in the program than the symbol table can handle.
SY	1,2	Illegal symbol: A label field contains an illegal character or is greater than 5 characters. A label with illegal characters may result in an erroneous assembly if not corrected. A long label is truncated to the left-most 5 character.
SY	2	Illegal symbol: A symbolic term in the operand field is greater than five characters; the symbol is truncated to the left-most 5 characters.
UN	1,2	Undefined symbol: a) A symbolic term in an operand field is not defined in the label field of an instruction. b) A symbol appearing in the operand field of one of the following pseudo operations was not defined previously in the source program: BSS ASC EQU ORG

APPENDIX

BINARY LOADERS

There are two basic approaches to loading a binary object program into memory.

The first (and the simplest and most primitive) way is to imitate the basic binary loader for the 2100-series computers. With this approach there is a 30 to 50 word program that must be resident in some unused portion of the BPC system's memory. This program performs the necessary input activity while understanding the format of the binary tape. There are several things to note about this approach:

1. The binary loader itself can only be loaded by hand - a tedious and error-prone activity. This is an especially grievous drawback if no non-volatile memory is available to contain the loader.
2. It is possible that the system under development might eventually not have room in memory for a resident loader.
3. If the system does not have an IOC, a special interface to the IDA bus is necessary. These come in two flavors:

   a. Build a special interface that acts like a memory address. It can be set to respond to an unused register address (very easy if RAL is used) or to a non-existent or non-decoded main-memory address. To load a byte in A from a photo reader whose interface thinks it is location 30_8, the loader would do a LDA 30B. The interface recognizes the memory address as its own, starts the photo reader and gets the byte, and holds the byte on the IDA bus, giving Memory Complete only when all photo reader activity is complete. In this way no special handshake is required, and to read a word from the tape it is necessary only to:

```
LDA    30B
SAL     8
IOR    30B
```

   b. Use a Model 30, or other calculator, programmed to read the data from the photo reader. The calculator sends the data to the IDA bus through an 11202-BIB combination (slightly supplemented) - all of which are off-the-shelf components. This allows a somewhat simpler interface and also a simpler resident binary loader: the check-sum can be checked and then removed from the instruction-word-stream by the program in the calculator.
   The 11202-BIB combination must be supplemented with memory address decoding; however the existing Flag convention can take the place of the missing Memory Complete circuitry. The resident binary loader still addresses memory to get a byte from the reader:

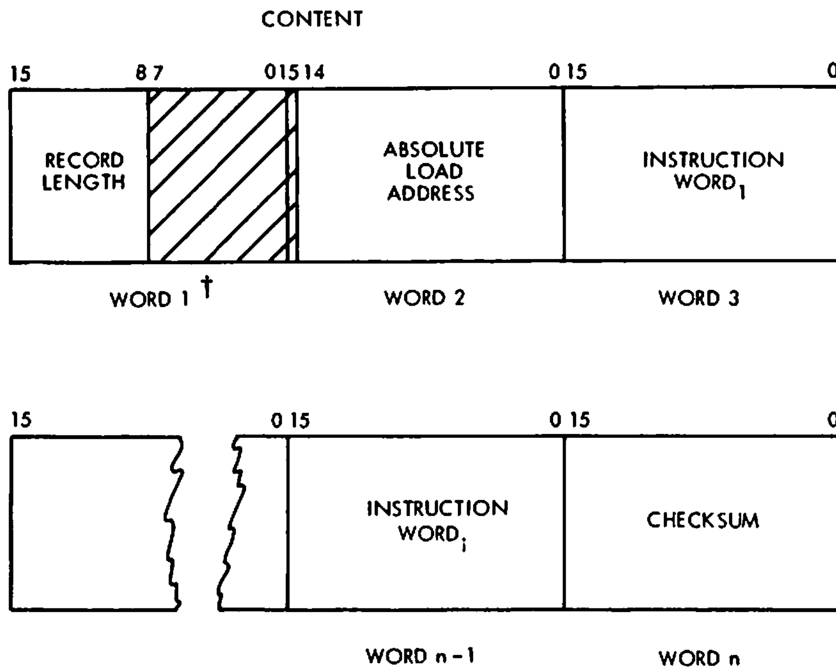
```
LDBYT    SFC    LDBYT
         LDA    I/OAD,I
         SAL    8
         SFC    *
         IOR    I/OAD,I
          .
          .
I/OAD    DEF    XXX ADDRESS DECODED BY INTERFACE
```

BINARY LOADERS (CONT.)

The second general approach is much more sophisticated, but is a lot easier. It is to use the ET-8332. The ET-8332 is much more than just a loader; it is that in addition to being a full-scale test apparatus for controlling traffic on the IDA bus and debugging software. It is controlled by software executed by a Model 30, and has many useful features. As far as loading is concerned, no resident loader is required in the memory of the BPC system under development, and object code can be stored on a disc. The ET-8332 is generally considered superior to an ordinary single-step-tester.

OUTPUT PAPER TAPE FORMAT

ABSOLUTE BINARY OBJECT PROGRAM



EXPLANATION

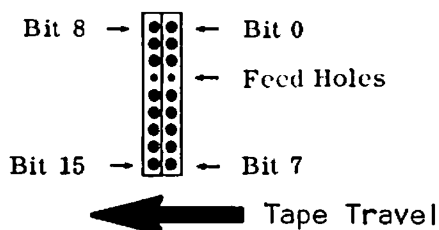
RECORD LENGTH = NUMBER OF WORDS IN RECORD EXCLUDING WORDS 1 AND 2 AND THE LAST WORD.

ABSOLUTE LOAD ADDRESS: STARTING ADDRESS FOR LOADING THE INSTRUCTIONS WHICH FOLLOW

INSTRUCTION WORDS: ABSOLUTE INSTRUCTIONS OR DATA

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS EXCEPT FIRST AND LAST

†Each word represents two frames arranged as follows:



ADDING PRE-DEFINED SYMBOLS TO ASMA

It is a relatively easy task to add pre-defined symbols to ASMA. What is necessary is the creation of some extra source text for ASMAl and ASMA4. Both must be changed; whatever modification made to one must also be made to the other. After modification, these segments must be re-assembled, and the entire program collection re-loaded.

Below is a partial source listing of ASMA1 and ASMA4, in the vicinity of lines 415-430. (The exact location in each keeps changing over time. I give up trying to keep this page accurate).

APPROXIMATE LINE NUMBERS

```
0411  COUNT DEC 58
0412  PRELD OCT 20101,0      A REG = 0
0413        OCT 20102,1      B REG = 1
0414        OCT 20120,2      P REG = 2
0415        OCT 20122,3      R REG = 3
0416        OCT 30122,32040,4     R4=4
0417        OCT 30122,32440,5     R5=5
0418        OCT 30122,33040,6       R6=6
0419        OCT 30122,33440,7       R7=7
0420        OCT 30111,53040,10          IV
0421        OCT 30120,40440,11          PA
0422        OCT 20127,12                W
0423        OCT 40104,46501,50101,13    DMAPA
0424        OCT 40104,46501,46501,14    DMAMA
0425        OCT 40104,46501,41440,15    DMAC
0426        OCT 20103,16                C
0427        OCT 20104,17                D
0428  AR1AD OCT 30101,51061,77770       AR1
0429        OCT 30101,51062,20          AR2 = 20
0430        OCT 30123,42440,24          SE - 24
0431        OCT 0,0,0,0,0    DUMMY END OF SYMBOL TABLE
**** LIST END ****
```

Here is how to add a pre-defined symbol:

1. The symbol to be added must, in every way, conform to the rules for labels and their permissible values.
2. If the symbol has an even number of characters, imagine that it has a trailing blank (ƀ) as the right most character, so that the "number of characters" is always odd.
3. Using the ASCII conversion table in this appendix, convert the symbol into one or more octal integers. Note how the left-most character is right-justified into an all-zero word.

DOGG

DOGG ƀ

```
all zeros = 000000
        D =+000104
           -------
            000104
```

ADDING PRE-DEFINED SYMBOLS TO ASMA

3. (cont.)

```
O =  047400
G = +000107
     ------
     047507

G =  043400
b̸ = +000040
     ------
     043440
```

4. So far we have the sequence:

   000104, 047507, 043440

   The next step is to add one more word, representing the octal value of the symbol. Suppose DOGG is to equal 77B. Then this generates the sequence:

   000104, 047507, 043440, 000077

5. Count the number of words (in this case 4). Insert this number into the first word exactly as shown below:

   040104, 047507, 043440, 000077
     ↑

6. Create an OCT statement that will generate the same sequence of words:

   OCT 40104,47507,43440,77

   Note that leading zeros may be omitted.

7. One other change in the program source text is necessary: The value of the word called COUNT must be changed (line 398 in ASMA4). COUNT is the total number of words in the symbol table pre-load.

   In our example, we are adding four words. So COUNT would change from its base value of 58_{10} to:

   COUNT DEC 62

8. Prepare edits that will change COUNT to its new value in both ASMA1 and ASMA4, and that will insert the new octal constants between lines 418 and 419 of ASMA1 and between lines 417 and 418 of ASMA4.

9. Make the edits, re-assemble, and re-load.

10. You can verify proper behavior of the symbol table pre-load, as well as obtain a complete list of the pre-loaded symbols, by assembling any program including a T in its Control Statement.

ADDING PRE-DEFINED SYMBOLS TO ASMA (CONT.)

The symbols that are pre-defined by the assembler are shown below. With the exception of ARI, all these symbols refer to registers within the various elements of the system.

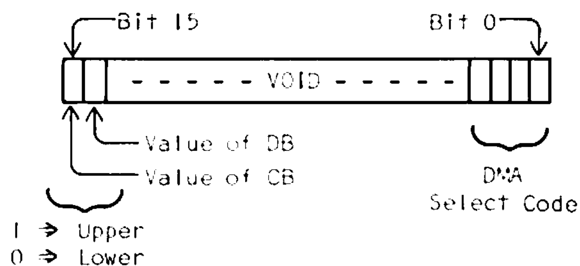
REGISTERS & ASMA PRE-DEFINED SYMBOLS

Octal Address	Name	Location	Description (# of Bits)
0	A	BPC	Arithmetic Accumulator (16)
1	B	BPC	Arithmetic Accumulator (16)
2	P	BPC	Program Location Counter (least 15 of 16 or 16)
3	R	BPC	Return Stack Pointer (least 15 of 16 or 16)
4	R4	IOC	Peripheral Activity Designator (—)
5	R5	IOC	Peripheral Activity Designation (—)
6	R6	IOC	Peripheral Activity Designator (—)
7	R7	IOC	Peripheral Activity Designator (—)
10	IV	IOC	Interrupt Vector (upper 12 of 16)
* → 11	PA	IOC	Peripheral Address Register (least 4 of 16)
12	W	IOC	Working Register (16)
† → 13	DMAPA	IOC	2 MSB = CB & DB; 4 LSB = DMA Periph. Add. Reg.
14	DMAMA	IOC	DMA Memory Address & Direction Register (16)
15	DMAC	IOC	DMA Count Register (16)
16	C	IOC	Stack Pointer (16)
17	D	IOC	Stack Pointer (16)
20-23	AR2	EMC	BCD Arithmetic Accumulator (4 x 16)
24	SE	EMC	Shift Extend Register (least 4 of 16)
* → 25-27	X	EMC	Internal Arithmetic Register (3 X 16)
30-37	UNASSIGNED		
77770/ 177770	ARI	R/W	BCD Arithmetic Register (4 x 16)

* Not available for general use. Part of processes internal to a chip. It is best to pretend that these registers do not exist.

† Read register 13_8 produces:

CB and DB are actually discrete registers, and while they can only be read by reading R13, storinging into R13 will not alter their values. Use the CBL, CBU, DBL and DBU machine instructions for that purpose. CB and DB exist in the 16-bit version only.



THE STRUCTURE OF THE ASSEMBLER

The assembler is a segmented program that can run under either DOS-M or RTE. The names of the segments are:

ASMA	the main segment
ASMAD	overlay segment
ASMA1	overlay segment
ASMA2	overlay segment
ASMA4	overlay segment
ASMA5	overlay segment

Note that there is no ASMA3. Special procedures are required when loading segmented programs; see the operating manual for your system.

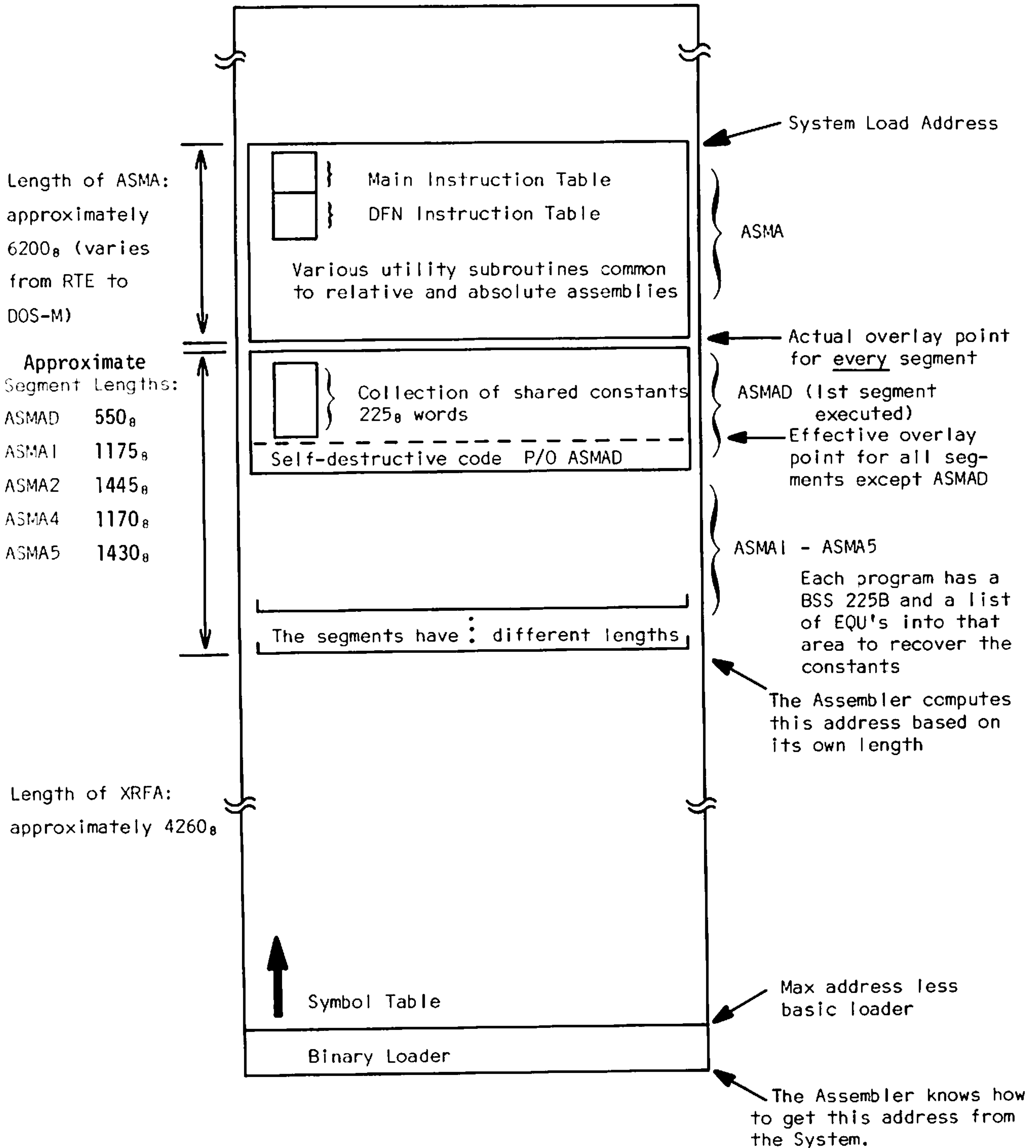
The differences between the DOS-M version and the RTE version is entirely contained within ASMA (main segment). Whether ASMA is for DOS-M or for RTE is controlled at the time ASMA itself is assembled (by ASMB, the regular assembler). It is merely a matter of an N or a Z in the Control Statement of the source for ASMA (main segment only). This is fully explained by the comments in the listing.

The illustration on the next page is a pictorical representation of ASMA when it is in core.

APPENDIX

Order of execution:

Relative	Non-Relative ("Absolute")
ASMA	ASMA
ASMAD	ASMAD
ASMA1 (uses ASMA)	ASMA4 (uses ASMA)
ASMA2 (uses ASMA)	ASMA5 (uses ASMA)



APPENDIX

PSEUDO INSTRUCTIONS

ABS m

Defines a 16-bit value to be stored at the location represented by the label. (ASMA: Assembler-19)

ASC n, < 2n characters >

Converts a string of 2n alphanumeric characters in ASCII code into n consecutive words. (ASMA: Assembler-20)

BSS m

Advances the program location counter according to the value of the operand. (ASMA: Assembler-23)

CLA

Clear A. The assembler turns this mnemonic into an SAR 16 (shift A right 16). This has the effect of clearing the A register. (BPC: Instructions-4)

CLB

Clear B. Similar to CLA. (BPC: Instructions-4)

DEC $d_1[,d_2,....,d_n]$

Records a string of integer decimal constants into consecutive words. (ASMA: Assembler-21)

DEF m [,I]

Generates one word of memory as a 15-bit or 16-bit address which may be used as the object of an indirect address found elsewhere in the source program. (AMSA: Assembler-18)

DFN < mnemonic >, < type >, < bit pattern >

Defines a machine instruction with the given 3-character mnemonic. (ASMA: Assembler-13)

END

Terminates the program; marks the physical end of the source language statements. (ASMA: Assembler-17)

EQU m

Assigns to a symbol a value other than the one normally assigned by the program location counter. (ASMA: Assembler-19)

HED < heading >

Allows the programmer to specify a heading to be printed at the top of each page of the source program listing. (ASMA: Assembler-25)

IFN

Source language statements after IFN and before the next XIF are included in the program if the character "N" is specified in the ASMB control statement. (ASMA: Assembler-15)

IFZ

Source language statements after the IFZ and before the next XIF pseudo instructions are included in the program if the character "Z" is specified in the ASMB control statement. (ASMA: Assembler-15)

LST

Causes the source program listing, terminated by a UNL, to be resumed. (ASMA: Assembler-24)

NOP

Null operation. The assembler turns this mnemonic into a LDA A. (BPC: Instructions-4)

OCT $O_1[,O_2,....,O_n]$

Stores one or more integer octal constants in consecutive words of the object program. (ASMA: Assembler-22)

ORG m

Defines the origin of a program, or the origins of subsequent sections of programming. (ASMA: Assembler-11)

ORR

Automatic reset of the value of the program location counter. (ASMA: Assembler-11)

PSEUDO INSTRUCTIONS (CONT.)

REP n

Causes the repetition of the next statement a specified number of times.
(ASMA: Assembler-17)

SKP

Causes the source program listing to be skipped to the top of the next page.
(ASMA: Assembler-24)

SPC n

Causes the source program listing to be skipped a specified number of lines.
(ASMA: Assembler-24)

SUP

Suppresses the output of additional code lines from the source program listing.
(ASMA: Assembler-24)

UNL

Output is suppressed from the assembly listing for all subsequent instructions and comments until either an LST or END is encountered.
(ASMA: Assembler-23)

UNS

Causes the printing of additional coding lines, terminated by a SUP, to be resumed.
(ASMA: Assembler-24)

XIF

Terminates conditional assembly text.
(ASMA: Assembler-15)

$$$

Causes any as yet un-outputted binary to be properly outputted as a complete record.
(ASMA: Assembler-15)

MACHINE INSTRUCTIONS

AAR n

Arithmetic right shift of A. A is shifted right n places with the sign bit (bit 15) filling all vacated bit positions. (BPC: Instructions-5)

ABR n

Arithmetic right shift of B. B is shifted right n places with the sign bit (bit 15) filling all vacated bit positions. (BPC: Instructions-5)

ADA m [,I]

Add the contents of m to A.
(BPC: Instructions-2)

ADB m [,I]

Add the contents of m to B.
(BPC: Instructions-2)

AND m [,I]

Logical "and" of A and m; the result is left in A. (BPC: Instructions-3)

CBL

C Block Lower. Clears the CB register. 16-bit IOC only. (IOC: Instructions-13)

CBU

C Block Upper. Sets the CB register. 16-bit IOC only. (IOC: Instructions-13)

CDC

Clear Decimal Carry. (EMC: Instructions-18)

CLR N

Clear N words. This instructions clears 1-16 consecutive words, beginning with location < A >.
(EMC: Instructions-16)

CMA

Complement A. The A register is replaced by its one's (bit by bit) complement.
(BPC: Instructions-10)

APPENDIX

MACHINE INSTRUCTIONS (CONT.)

CMB

Complement B. The B register is replaced by its one's (bit by bit) complement. (BPC: Instructions-10)

CMX

Ten's complement of AR1. 15-bit version has a DMA-related bug. (EMC: Instructions-17)

CMY

Ten's complement of AR2. (EMC: Instructions-18)

CPA m [,I]

Compare the contents of m with the contents of A; skip if unequal. (BPC: Instructions-2)

CPB m [,I]

Compare the contents of m with the contents of B; skip if unequal. (BPC: Instructions-2)

DBL

D Block Lower. Clears the DB register. 16-bit IOC only. (IOC: Instructions-13)

DBU

D Block Upper. Sets the DB register. 16-bit IOC only. (IOC: Instructions-13)

DDR

Disable Data Request. Cancels the DMA Mode and the Pulse Count Mode. 15-bit version has DMA-related bug; DDR is usable in the 16-bit version only. (IOC: Instructions-15)

DIR

Disable the interrupt system, cancels EIR. (IOC: Instructions-14)

DMA

Enable the DMA mode. Cancels PCM and DDR. (IOC: Instructions-15)

DRS

Mantissa right shift of AR1 one time. (EMC: Instructions-17)

DSZ m [,I]

Decrement m; then skip if zero. (BPC: Instructions-3)

EIR

Enable the interrupt system. (IOC: Instructions-14)

EXE $0 \leq m \leq 37_8$ [,I]

Execute register m. The contents of any register can be treated as the current instruction, and executed in the normal manner. The next instruction executed will be the one following the EXE m, unless the code in m causes a branch. 15-bit version has minor bug related to interrupt. (BPC: Instructions-11)

FDV

Fast Divide. The mantissas of AR1 and AR2 are added together until the first decimal overflow occurs. The result of these additions accumulates in AR2. (EMC: Instructions-19)

FMP

Fast Multiply. The mantissas of AR1 and AR2 are added together (along with DC as D_{12}) $< B_{0-3} >$-times; the result accumulates in AR2. (EMC: Instructions-19)

FXA

Fixed-point addition. The mantissas of AR1 and AR2 are added together, and the result is left in AR2. (EMC: Instructions-18)

IOR m [,I]

Inclusive (ordinary) "or" of A and m; the result is left in A. (BPC: Instructions-3)

ISZ m [,I]

Increment m; then skip if zero. (BPC: Instructions-3)

JMP m [,I]

Jump to m. Program execution continues at location m. (BPC: Instructions-3)

MACHINE INSTRUCTIONS (CONT.)

JSM m [,I]

Jump to subroutine. The contents of the return stack register (R) are incremented by one and the contents of P stored in R,I. Program execution resumes at m. (BPC: Instructions-3)

LDA m [,I]

Load A from m. (BPC: Instructions-2)

LDB m [,I]

Load B from m. (BPC: Instructions-2)

<mem. ref. inst.> <reg. 4-7> [,I]

Initiate an I/O Bus Cycle. Memory reference instructions 'reading' from reg. cause input I/O Bus Cycles; those 'writing' to reg. cause output I/O Bus Cycles. In either case the exchange is between A or B and the interface addressed by the PA register (Peripheral Address Register - 11_8). (IOC: Instructions-14)

MLY

Mantissa left shift of AR2 one time. (EMC: Instructions-17)

MPY

Binary Multiply Using Booth's Algorithm. (EMC: Instructions-19)

MRX

Mantissa right shift of AR1 < B_{0-3} >-times. (EMC: Instructions-16)

MRY

Mantissa right shift of AR2 < B_{0-3} >-times. (EMC: Instructions-17)

MWA

Mantissa Word Add. < B > is taken as four BCD digits, and added, as D_9 through D_{12}, to AR2. DC is also added in as a D_{12}. The result is left in AR2. (EMC: Instructions-18)

NRM

Normalize AR2. The mantissa digits of AR2 are shifted left until $D_1 \neq 0$. (EMC: Instructions-17)

PBC reg. 0-7 [,I/,D]

Place the right half of reg. into the stack pointed at by C. (IOC: Instructions-12)

PBD reg. 0-7 [,I/,D]

Place the right half of reg. into the stack pointed at by D. (IOC: Instructions-12)

PCM

Enable the Pulse Count Mode. (IOC: Instructions-15)

PWC reg. 0-7 [,I/,D]

Place the entire word of reg. into the stack pointed at by C. (IOC: Instructions-12)

PWD reg. 0-7 [,I/,D]

Place the entire word of reg. into the stack pointed at by D. (IOC: Instructions-12)

RAR n

Rotate A right. A is rotated right n places, with bit 0 rotating into bit 15. (BPC: Instructions-5)

RBR n

Rotate B right. B is rotate right n places, with bit 0 rotating into bit 15. (BPC: Instructions-5)

RET n [,P]

Return. A read R,I occurs. That produces the address (< P >) of the latest JSM that occurred. The BPC then jumps to address < P > + n. The value of n may range from -32 to 31, inclusive. At the conclusion of the RET R is decremented by one.
The ordinary, everyday, return is RET 1. If a P is present, it "pops" the interrupt system. (BPC: Instructions-3)

MACHINE INSTRUCTIONS (CONT.)

RIA * ± n/m
Skip if A is not zero, then increment A.
(BPC: Instructions-7)

RIB * ± n/m
Skip if B is not zero, then increment B.
(BPC: Instructions-7)

RLA * ± n/m [,S/,C]
Skip if the least significant bit of A is non-zero. If either S or C is present, bit 0 is altered accordingly after the test.
(BPC: Instructions-9)

RLB * ± n/m [,S/,C]
Skip if the least significant bit of B is non-zero. If either S or C is present, bit 0 is altered accordingly after the test.
(BPC: Instructions-9)

RZA * ± n/m
Skip if A not zero. (BPC: Instructions-7)

RZB * ± n/m
Skip if B not zero. (BPC: Instructions-7)

SAL n
Shift A left. A is shifted left n places with all vacated bit positions cleared.
(BPC: Instructions-5)

SAM * ± n/m [,S/,C]
Skip if A minus. If either S or C is present, bit 15 is altered accordingly after the test.
(BPC: Instructions-9)

SAP * ± n/m [,S/,C]
Skip if A positive. If either S or C is present, bit 15 is altered accordingly after the test.
(BPC: Instructions-9)

SAR n
Shift A right. A is shifted right n places with all vacated bit positions cleared.
(BPC: Instructions-5)

SBL n
Shift B left. B is shifted left n places with all vacated bit positions cleared.
(BPC: Instructions-5)

SBM * ± n/m [,S/,C]
Skip if B minus. If either S or C is present, bit 15 is altered accordingly after the test.
(BPC: Instructions-9)

SBP * ± n/m [,S/,C]
Skip if B positive. If either S or C is present, bit 15 is altered accordingly after the test. (BPC: Instructions-9)

SBR n
Shift B right. B is shifted right n places with all vacated bit positions cleared.
(BPC: Instructions-5)

SDC * ± n/m
Skip if decimal carry clear.
(BPC: Instructions-8)

SDI
Set DMA Inwards. 16-bit IOC instruction that sets the direction of DMA transfers to be from the peripheral to the memory.
(IOC: Instructions-15)

SDO
Set DMA Outwards. 16-bit IOC instruction that sets the direction of DMA transfers to be from the memory to the peripheral.
(IOC: Instructions-15)

SDS * ± n/m
Skip if decimal carry set.
(BPC: Instructions-8)

SEC * ± n/m [,S/,C]
Skip if extend clear. If either S or C is present, E is altered accordingly after the test.
(BPC: Instructions-10)

MACHINE INSTRUCTIONS (CONT.)

SES * ± n/m [,S/,C]
Skip if extend set. If either S or C is present, E is altered accordingly after the test. (BPC: Instructions-10)

SFC * ± n/m
Skip if Flag line clear. (BPC: Instructions-8)

SFS * ± n/m
Skip if Flag line set. (BPC: Instructions-8)

SHC * ± n/m
Skip if Halt line clear. (BPC: Instructions-8)

SHS * ± n/m
Skip if Halt line set. (BPC: Instructions-8)

SIA * ± n/m
Skip if A is zero, then increment A. (BPC: Instructions-7)

SIB * ± n/m
Skip if B is zero, then increment B. (BPC: Instructions-7)

SLA * ± n/m [,S/,C]
Skip if the least significant bit of A is zero. If either S or C is present, bit 0 is altered accordingly after the test. (BPC: Instructions-8)

SLB * ± n/m [,S/,C]
Skip if the least significant bit of B is zero. If either S or C is present, bit 0 is altered accordingly after the test. (BPC: Instructions-9)

SOC * ± n/m [,S/,C]
Skip if overflow clear. If either S or C is present, the OV register is altered accordingly after the test. (BPC: Instructions-10)

SOS * ± n/m [,S/,C]
Skip if overflow set. If either S or C is present, the OV register is altered accordingly after the test. (BPC: Instructions-9)

SSC * ± n/m
Skip if Status line clear. (BPC: Instructions-8)

SSS * ± n/m
Skip if Status line set. (BPC: Instructions-8)

STA m [,I]
Store the contents of A in m. (BPC: Instructions-2)

<stack inst.> <reg. 4-7> [,I/,D]
Initiate an I/O Bus Cycle. Place instructions 'read' from reg., therefore they cause input I/O Bus Cycles. Withdraw instructions 'write' into reg., therefore they cause output I/O Bus Cycles. In either case the exchange is between the addressed stack location and the interface addressed by PA. (IOC: Instructions-14)

STB m [,I]
Store the contents of B in m. (BPC: Instructions-3)

SZA * ± n/m
Skip if A zero. (BPC: Instructions-6)

SZB * ± n/m
Skip if B zero. (BPC: Instructions-7)

TCA
Two's complement A. The A register is replaced by its one's (bit by bit) complement, and then incremented by one. (BPC: Instructions-10)

TCB
Two's complement B. The B register is replaced by its one's (bit by bit) complement, and then incremented by one. (BPC: Instructions-10)

MACHINE INSTRUCTIONS (CONT.)

WBC reg. 0-7 [,I/,D]
Withdraw a byte from the stack pointed at by C, and put it into the right half of reg.
(IOC: Instructions-13)

WBD reg. 0-7 [,I/,D]
Withdraw a byte from the stack pointed at by D, and put it into the right half of reg.
(IOC: Instructions-13)

WWC reg. 0-7 [,I/,D]
Withdraw an entire word from the stack pointed at by C, and put it into reg.
(IOC: Instructions-13)

WWD reg. 0-7 [,I/,D]
Withdraw an entire word from the stack pointed at by D, and put it into reg.
(IOC: Instructions-13)

XFR N
Transfer N words. This instruction transfers the 1-16 (N) consecutive words beginning at location < A > to those beginning at < B >.
(EMC: Instructions-16)

INSTRUCTIONS BIT PATTERNS

GROUP: MEMORY REFERENCE (ASMA TYPE 16)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDA	D/I	0	0	0	0	B/C										
LDB	D/I	0	0	0	1	B/C										
CPA	D/I	0	0	1	0	B/C										
CPB	D/I	0	0	1	1	B/C										
ADA	D/I	0	1	0	0	B/C										
ADB	D/I	0	1	0	1	B/C										
STA	D/I	0	1	1	0	B/C										
STB	D/I	0	1	1	1	B/C										
JSM	D/I	1	0	0	0	B/C										
ISZ	D/I	1	0	0	1	B/C										
AND	D/I	1	0	1	0	B/C										
DSZ	D/I	1	0	1	1	B/C										
IOR	D/I	1	1	0	0	B/C										
JMP	D/I	1	1	0	1	B/C										

* 10 BIT ADDRESS FIELD.
* ADDRESSES $0\text{-}37_8$ ARE REGISTERS.
* FOR BIT 9=0, BITS 0-8 = POSITIVE ADDR.
* FOR BIT 9=1, ADDRESS IS NEGATIVE. IGNORE BIT 9, COMPLEMENT BITS 0-8, THEN ADD ONE.
* BASE PAGE ADDRESS ENCODING IS ALWAYS WITH RESPECT TO MEMORY LOCATION ZERO.
* CURRENT PAGE ENCODING:

  (ABSOLUTE) RELATIVE TO THE MIDDLE OF THE PAGE (1000B, 3000B, ETC.)

  (RELATIVE) RELATIVE TO THE CURRENT VALUE OF P, +511, -512.

D/I (DIRECT/INDIRECT) AND B/C (BASE PAGE/CURRENT PAGE) ARE CODED AS 0/1.

GROUP: SHIFT-ROTATE (ASMA TYPE 27)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AAR	1	1	1	1	0	0	0	1	0	0	0	0				
ABR	1	1	1	1	1	0	0	1	0	0	0	0				
SAR	1	1	1	1	0	0	0	1	0	1	0	0				
SBR	1	1	1	1	1	0	0	1	0	1	0	0				
SAL	1	1	1	1	0	0	0	1	1	0	0	0				
SBL	1	1	1	1	1	0	0	1	1	0	0	0				
RAR	1	1	1	1	0	0	0	1	1	1	0	0				
RBR	1	1	1	1	1	0	0	1	1	1	0	0				

* 4 BITS OF SHIFT-ROTATE FIELD.
* IN SOURCE $1 \le N \le 16$.
* BINARY IN THIS FIELD IS N-1.

INSTRUCTIONS BIT PATTERNS (CONT.)

GROUP: SKIP (ASMA TYPE 25)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RZA	0	1	1	1	0	1	0	0	0	0	*					
RZB	0	1	1	1	1	1	0	0	0	0						
SZA	0	1	1	1	0	1	0	1	0	0						
SZB	0	1	1	1	1	1	0	1	0	0						
RIA	0	1	1	1	0	1	0	0	0	1						
RIB	0	1	1	1	1	1	0	0	0	1						
SIA	0	1	1	1	0	1	0	1	0	1						
SIB	0	1	1	1	1	1	0	1	0	1						
SFS	0	1	1	1	0	1	0	0	1	0						
SFC	0	1	1	1	0	1	0	1	1	0						
SSS	0	1	1	1	1	1	0	0	1	0						
SSC	0	1	1	1	1	1	0	1	1	0						
SDS	0	1	1	1	0	1	0	0	1	1						
SDC	0	1	1	1	0	1	0	1	1	1						
SHS	0	1	1	1	1	1	0	0	1	1						
SHC	0	1	1	1	1	1	0	1	1	1						

* 6 BIT SKIP FIELD, +31, -32.

* IF BIT 5=0, SKIP TO P+#; #=BITS 0 THRU 4.

* IF BIT 5=1, SKIP TO P-#; #=1+ COMP OF BITS 0-4.

GROUP: RETURN (ASMA TYPE 42)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RET	1	1	1	1	0	0	0	0	1	$\overline{P}$/P	6 BIT, 2'S COMPLEMENT SKIP FIELD (ALLOWS -32 THRU +31).					

$\overline{P}$/P (DON'T POP/POP THE IOC) ENCODED AS 0/1.

INSTRUCTIONS BIT PATTERNS (CONT.)

GROUP: COMPLEMENT (ASMA TYPE 30)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMA	1	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0
CMB	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0
TCA	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0
TCB	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0

GROUP: ALTER (ASMA TYPE 53)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RLA	0	1	1	1	0	1	1	1	H/$\overline{H}$	C/S						
RLB	0	1	1	1	1	1	1	1	H/$\overline{H}$	C/S						
SLA	0	1	1	1	0	1	1	0	H/$\overline{H}$	C/S						
SLB	0	1	1	1	1	1	1	0	H/$\overline{H}$	C/S						
SAP	1	1	1	1	0	1	0	0	H/$\overline{H}$	C/S						
SBP	1	1	1	1	1	1	0	0	H/$\overline{H}$	C/S						
SAM	1	1	1	1	0	1	0	1	H/$\overline{H}$	C/S						
SBM	1	1	1	1	1	1	0	1	H/$\overline{H}$	C/S						
SOC	1	1	1	1	0	1	1	0	H/$\overline{H}$	C/S						
SOS	1	1	1	1	0	1	1	1	H/$\overline{H}$	C/S						
SEC	1	1	1	1	1	1	1	0	H/$\overline{H}$	C/S						
SES	1	1	1	1	1	1	1	1	H/$\overline{H}$	C/S						

* 6 BIT SKIP FIELD, +31, -32.
* IF BIT 5=0, SKIP TO P+#; #=BITS 0 THRU 4.
* IF BIT 5=1, SKIP TO P-#, #=1+ COMP OF BITS 0-4.

H/$\overline{H}$ (HOLD/DON'T HOLD) AND C/S (CLEAR/SET) ARE CODED AS 0/1.

HOWEVER: H/$\overline{H}$ IS SET BY THE ASSEMBLER ITSELF. IF NEITHER S NOR C IS PRESENT, BOTH H/$\overline{H}$ AND C/S ARE MADE 0'S. THE PRESENCE OF EITHER A C OR AN S PRODUCES H (A 1).

INSTRUCTION BIT PATTERNS (CONT.)

GROUP: EXECUTE (ASMA TYPE 41)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXE	D/I	1	1	1	0	0	0	0	0	0	0	5 BIT REGISTER ADDRESS ($0-37_8$).				

D/I (DIRECT/INDIRECT) ENCODED AS 0/1.

GROUP: 16-BIT IOC ONLY (ASMA TYPE 46)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SDO	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0
SDI	0	1	1	1	0	0	0	1	0	0	0	0	1	0	0	0
DBL	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0
CBL	0	1	1	1	0	0	0	1	0	1	0	0	1	0	0	0
DBU	0	1	1	1	0	0	0	1	0	1	0	1	0	0	0	0
CBU	0	1	1	1	0	0	0	1	0	1	0	1	1	0	0	0

APPENDIX

INSTRUCTION BIT PATTERNS (CONT.)

GROUP: STACK (ASMA TYPE 43)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PWC	0	1	1	1	0	0	0	1	I/D	1	1	0	0	*		
PBC	0	1	1	1	1	0	0	1	I/D	1	1	0	0	*		
PWD	0	1	1	1	0	0	0	1	I/D	1	1	0	1	*		
PBD	0	1	1	1	1	0	0	1	I/D	1	1	0	1	*		
WWC	0	1	1	1	0	0	0	1	I/D	1	1	1	0	*		
WBC	0	1	1	1	1	0	0	1	I/D	1	1	1	0	*		
WWD	0	1	1	1	0	0	0	1	I/D	1	1	1	1	*		
WBD	0	1	1	1	1	0	0	1	I/D	1	1	1	1	*		

* 3 BIT REGISTER ADDRESS FIELD $(0\text{-}7_8)$.

* PLACE INST'S INC/DEC THE STACK POINTER BEFORE THE OPERATION.

* WITHDRAW INST'S INC/DEC THE STACK POINTER AFTERWARDS.

1. I/D (INCREMENT/DECREMENT) IS ENCODED AS 0/1
2. THE ASSEMBLER DEFAULTS TO INCREMENT FOR PLACE INSTRUCTIONS, AND TO DECREMENT FOR WITHDRAW INSTRUCTIONS.
3. FOR 15-BIT/16-BIT BYTE INSTRUCTIONS, A 1 IN BIT 15/0 OF THE POINTER REGISTER IMPLIES A LEFT-HALF

GROUP: INTERRUPT (ASMA TYPE 30)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EIR	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0
DIR	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0

GROUP: DMA (ASMA TYPE 30)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0
PCM	0	1	1	1	0	0	0	1	0	0	1	0	1	0	0	0
DDR	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0

INSTRUCTION BIT PATTERNS (CONT.)

GROUP: FOUR WORD OPERATION (ASMA TYPE 27)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLR	0	1	1	1	0	0	1	1	1	0	0	0	* 4 BIT FIELD # OF WORDS * BINARY = N-1			
XFR	0	1	1	1	0	0	1	1	0	0	0	0				

GROUP: MANTISSA SHIFT (ASMA TYPE 30)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MRX	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0
DRS	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1
MLY	0	1	1	1	1	0	1	1	0	1	1	0	0	0	0	1
MRY	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0
NRM	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0

GROUP: ARITHMETIC (ASMA TYPE 30)

INST. NAME	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FXA	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0
MWA	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
CMX	0	1	1	1	0	0	1	0	0	1	1	0	0	0	0	0
CMY	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0
FMP	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
FDV	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1
MPY	0	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1
CDC	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0

APPENDIX

15/16 BIT BPC CONSOLIDATED CODING SHEET

Group	Mnemonic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MEMORY REFERENCE	L D A/B	D/I	0	0	0	A/B	B/C	-10-BIT ADDRESS FIELD									
	C P A/B	↑	0	0	1	↑	↑										
	A D A/B		0	1	0	↓		-0-37₈ ARE REGISTERS									
	S T A/B		0	1	1	A/B		-IF BIT 9=0, BITS 0-8 ARE POSITIVE ADDRESS									
	J S M		1	0	0	0											
	I S Z		1	0	0	1		-IF BIT 9=1, BITS 0-8 ARE NEGATIVE ADDRESS;									
	A N D		1	0	1	0		COMPLEMENT BITS 0-8 THEN									
	D S Z		1	0	1	1		ADD 1									
	I O R		1	1	0	0	↓										
	J M P	↓	1	1	0	1	B/C										
	E X E	D/I	1	1	1	0	0	0	0	0	0	0	REGISTER ADD.				
SKIP	R Z A/B	0	1	1	1	A/B	1	0	0	0	0	-6-BIT SKIP FIELD +31,-32					
	R I A/B	0	1	1	1	↑	1	0	0	0	1						
	S Z A/B	0	1	1	1	↓	1	0	1	0	0	-IF BIT 5=0, SKIP TO (P+N), N= BITS 0-4					
	S I A/B	0	1	1	1	A/B	1	0	1	0	1						
	S F S/C	0	1	1	1	0	1	0	S/C	1	0						
	S D S/C	0	1	1	1	0	1	0	↑	1	1	-IF BIT 5=1, SKIP TO (P-N), N=1+ COMPLEMENT OF BITS 0-4					
	S S S/C	0	1	1	1	1	1	0	↓	1	0						
	S H S/C	0	1	1	1	1	1	0	S/C	1	1						
ALTER	S L A/B	0	1	1	1	A/B	1	1	0	H/H̄	C/S						
	R L A/B	0	1	1	1	↑	1	1	1	↑	↑						
	S A/B P	1	1	1	1	↓	1	0	0								
	S A/B M	1	1	1	1	A/B	1	0	1								
	S O C/S	1	1	1	1	0	1	1	C/S	↓	↓						
	S E C/S	1	1	1	1	1	1	1	C/S	H/H̄	C/S						
COMPLEMENT	T C A/B	1	1	1	1	A/B	0	0	0	0	0	1	0	0	0	0	0
	C M A/B	1	1	1	1	A/B	0	0	0	0	1	1	0	0	0	0	0
	R E T	1	1	1	1	0	0	0	0	1	P̄/P	2'S COMP. SKIP FIELD					
SHIFT-ROTATE	A A/B R	1	1	1	1	A/B	0	0	1	0	0	0	0	4-BIT SHIFT FIELD, N=1-16 IN SOURCE, BINARY=(N-1)			
	S A/B R	1	1	1	1	↑	0	0	1	0	1	0	0				
	S A/B L	1	1	1	1	↓	0	0	1	1	0	0	0				
	R A/B R	1	1	1	1	A/B	0	0	1	1	1	0	0				

—NOTES—

1. X/Y ALWAYS REPRESENTS 0/1
2. A/B DENOTES THE A OR B REGISTER
3. D/I DENOTES DIRECT OR INDIRECT
4. B/C DENOTES BASE PAGE OR CURRENT PAGE
5. P̄/P DENOTES DON'T POP OR POP THE IOC'S PA STACK
6. S/C DENOTES "SET" OR "CLEAR" IN AN INSTRUCTION MNEMONIC
7. H/H̄ DENOTES HOLD OR CHANGE THE TESTED BIT
8. C/S DENOTES CLEAR OR SET (,C OR ,S) THE TESTED BIT

15/16 BIT IOC CONSOLIDATED CODING SHEET

Group	Mnemonic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA*	S D O	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0
	S D I	0	1	1	1	0	0	0	1	0	0	0	0	1	0	0	0
INTERRUPT	E I R	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0
	D I R	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0
DMA	D M A	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0
	P C M	0	1	1	1	0	0	0	1	0	0	1	0	1	0	0	0
	D D R	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0
STACK*	D B L	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0
	C B L	0	1	1	1	0	0	0	1	0	1	0	0	1	0	0	0
	D B U	0	1	1	1	0	0	0	1	0	1	0	1	0	0	0	0
	C B U	0	1	1	1	0	0	0	1	0	1	0	1	1	0	0	0
STACK	P W/B C/D	0	1	1	1	W/B	0	0	1	I/D	1	1	0	C/D	3-BIT REG ADD		
	W W/B C/D	0	1	1	1	W/B	0	0	1	I/D	1	1	1	C/D			

—NOTES—

1. X/Y ALWAYS ENCODED AS 0/1
2. W/B DENOTES WORD/BYTE
3. I/D DENOTES INCREMENT/DECREMENT
4. C/D DENOTES THE C OR D REGISTERS
5. PLACE INST'S INC/DEC THE STACK POINTER BEFORE THE OPERATION
6. WITHDRAW INST'S INC/DEC THE STACK POINTER AFTERWARDS

* 16 BIT VERSION ONLY - ALL OTHERS ARE 15/16 BIT

15/16 BIT EMC CONSOLIDATED CODING SHEET

Group	Mnemonic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FOUR WORD	C L R	0	1	1	1	0	0	1	1	1	0	0	0	N=# OF WORDS BINARY=(N-1)			
	X F R	0	1	1	1	0	0	1	1	0	0	0	0				
MANTISSA SHIFT	M R X	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0
	D R S	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1
	M L Y	0	1	1	1	1	0	1	1	0	1	1	0	0	0	0	1
	M R Y	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0
	N R M	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0
ARITHMETIC	F X A	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0
	M W A	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
	C M X	0	1	1	1	0	0	1	0	0	1	1	0	0	0	0	0
	C M Y	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0
	F M P	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
	F D V	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1
	M P Y	0	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1
	C D C	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0

APPENDIX

HP CHARACTER SET

b_7				0	0	0	0	1	1	1	1
b_6				0	0	1	1	0	0	1	1
b_5				0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1								
0	0	0	0	NULL	DC_0	b̸	0	@	P	UNASSIGNED	UNASSIGNED
0	0	0	1	SOM	DC_1	!	1	A	Q		
0	0	1	0	EOA	DC_2	"	2	B	R		
0	0	1	1	EOM	DC_3	#	3	C	S		
0	1	0	0	EOT	DC_4 (STOP)	$	4	D	T		
0	1	0	1	WRU	ERR	%	5	E	U		
0	1	1	0	RU	SYNC	&	6	F	V		
0	1	1	1	BELL	LEM	' (APOS)	7	G	W		
1	0	0	0	FE_0	S_0	(	8	H	X		
1	0	0	1	HT / SK	S_1	)	9	I	Y		
1	0	1	0	LF	S_2	*	:	J	Z		
1	0	1	1	V_{TAB}	S_3	+	;	K	[		
1	1	0	0	FF	S_4	, (COMMA)	<	L	\		ACK
1	1	0	1	CR	S_5	—	=	M	]		Ⓘ
1	1	1	0	SO	S_6	.	>	N	↑		ESC
1	1	1	1	SI	S_7	/	?	O	←		DEL

Standard 7-bit set code positional order and notation are shown below with b_7 the high-order and b_1 the low-order, bit position.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1
EXAMPLE: The code for "R" is:	1	C	1	0	0	1	0

LEGEND

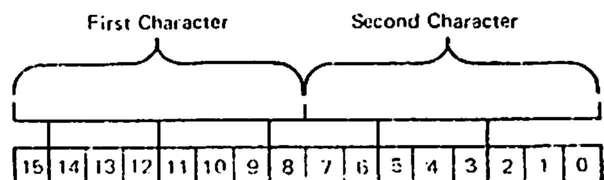
NULL	Null/Idle
SOM	Start of message
EOA	End of address
EOM	End of message
EOT	End of transmission
WRU	"Who are you?"
RU	"Are you...?"
BELL	Audible signal
FE_0	Format effector
HT	Horizontal tabulation
SK	Skip (punched card)
LF	Line feed
V_{TAB}	Vertical tabulation
FF	Form feed
CR	Carriage return
SO	Shift out
SI	Shift in
DC_0	Device control reserved for data link escape
DC_1-DC_3	Device Control
DC_4(Stop)	Device control (stop)
ERR	Error
SYNC	Synchronous idle
LEM	Logical end of media
S_0-S_7	Separator (information)
b̸	Word separator (space, normally non-printing)
<	Less than
>	Greater than
↑	Up arrow (Exponentiation)
←	Left arrow (Implies/Replaced by)
\	Reverse slant
ACK	Acknowledge
Ⓘ	Unassigned control
ESC	Escape
DEL	Delete/Idle

APPENDIX

CHARACTER CODES

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
A	040400	000101
B	041000	000102
C	041400	000103
D	042000	000104
E	042400	000105
F	043000	000106
G	043400	000107
H	044000	000110
I	044400	000111
J	045000	000112
K	045400	000113
L	046000	000114
M	046400	000115
N	047000	000116
O	047400	000117
P	050000	000120
Q	050400	000121
R	051000	000122
S	051400	000123
T	052000	000124
U	052400	000125
V	053000	000126
W	053400	000127
X	054000	000130
Y	054400	000131
Z	055000	000132
0	030000	000060
1	030400	000061
2	031000	000062
3	031400	000063
4	032000	000064
5	032400	000065
6	033000	000066
7	033400	000067
8	034000	000070
9	034400	000071
space	020000	000040
!	020400	000041
"	021000	000042
#	021400	000043
$	022000	000044
%	022400	000045
&	023000	000046
'	023400	000047
(	024000	000050
)	024400	000051
*	025000	000052
+	025400	000053
,	026000	000054
−	026400	000055
.	027000	000056
/	027400	000057

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
:	035000	000072
;	035400	000073
<	036000	000074
=	036400	000075
>	037000	000076
?	037400	000077
@	040000	000100
[	055400	000133
\	056000	000134
]	056400	000135
↑	057000	000136
←	057400	000137
ACK	036000	000174
①	036400	000175
ESC	037000	000176
DEL	037400	000177
NULL	000000	000000
SUM	000400	000001
EOA	001000	000002
EOM	001400	000003
EOT	002000	000004
WRU	002400	000005
RU	003000	000006
BELL	003400	000007
FE_0	004000	000010
HT/SK	004400	000011
LF	005000	000012
V_{TAB}	005400	000013
FF	006000	000014
CR	006400	000015
SO	007000	000016
SI	007400	000017
DC_0	010000	000020
DC_1	010400	000021
DC_2	011000	000022
DC_3	011400	000023
DC_4	012000	000024
ERR	012400	000025
SYNC	013000	000026
LEM	013400	000027
S_0	014000	000030
S_1	014400	000031
S_2	015000	000032
S_3	015400	000033
S_4	016000	000034
S_5	016400	000035
S_6	017000	000036
S_7	017400	000037



APPENDIX

BPC INSTRUCTION EXECUTION TIMES (IN CLOCK-TIMES)

INSTRUCTION	TIME FORMULA
LDA, LDB ADA, ADB AND, IOR	R(I + 2) + 1
CPA, CPB	R(I + 2) + 4
STA, STB	R(I + 1) + W + 1
ISZ, DSZ	R(I + 2) + W + 1
JMP	R(I + 1) + 2
JSM	R(I + 1) + W + 5
EXE	R(I + 1) + 2
RET	2R + 4
After-Skip Group	R + 8
Shift-Rotate Group	R + 3 + S
CMA, CMB TCA, TCB	R + 3

Where:

R = read-memory cycle time, expressed in BPC clock-times (must be an integer $\geq$ 4).
W = write-memory cycle time, expressed in BPC clock-times (must be an integer $\geq$ 4).
I = number of levels of indirect addressing (normally = 0).
S = number of positions to be shifted ($1 \leq S \leq 16$).

Note:

The read and write memory cycle times for a register located within the BPC, IOC, or EMC are 5 clock-times, unless such a reference is not a genuine register access; e.g., an I/O operation. In the latter case, it is simply however long it takes. (The 4 clock-time minimum is still effective however.)

APPENDIX

EMC INSTRUCTION EXECUTION TIMES (IN CLOCK-TIMES)

INSTRUCTION	TIME FORMULA	CONDITION
CLR	R + NW + 10	--
XFR	R(N + 1) + NW + 15	--
MRX	R + 20	If N = 0
	4R + 3W + 4B + 20	If N > 0
DRS	4R + 3W + 14	--
MLY	R + 26	--
MRY	R + 20	If N = 0
	R + 4B +27	If N > 0
NRM	R + Z + 17	If 0 ≤ N < 12
	R + 63	If N ≥ 12
FXA	4R + 16	--
MWA	R + 22	--
CMX	4R + 3W + 17	--
CMY	R + 17	--
FMP	R + 28	If B = 0
	4R + 13B + 18	If B > 0
FDV	4R + 13B + 13	--
MPY	R + 2T + 59	--
CDC	R + 5	--

Where:

R = read-memory cycle time, expressed in BPC clock-times (must be an integer ≥ 4).
W = write-memory cycle time, expressed in BPC clock-times (must be an integer ≥ 4).
N = bits 0-3 of the instruction word. (0 → 16)
Z = number of leading zeros in the mantissa of AR2.
B = bits of 0-3 of the B register contents.
T = number of 0-1 transitions plus the number of 1-0 transitions, in the A register, counting from an imaginary 0 just to the "right" of the LSB of A, to the MSB of A.

Note:

The read and write memory cycle times for register located within the system are the same as for the BPC.

IOC EXECUTION TIMES (IN CLOCK-TIMES)

REGISTER	CLOCK-TIMES
R_4 - R_7	7
R_8 - R_{15}	5

INSTRUCTION TIMES	TIME FORMULA
EIR, DIR, PCM, DMA, DDR, SDO, SDI, DBL, CBL, DBU, CBU	$R_M + 6$
PW*, PB*, WB*, WW*	$R_M + W_M + 11$
INTERRUPT	
Lockout (LI)	$LI_{Max} = E + 2$ $LI_{Min} = 2$
Execution	$R_R + R_M(I + 1) + W_M + 12$
DMA	
Lockout (LD)	$LD_{Min} = 2$ $LD_{Max} = 10$
Read	$LD + 3 + n(R_M + 4)$
Write	$LD + 3 + n(W_M + 3)$
PCT	$LD + 6n + 3$

Where:

R_M = read-memory cycle time in BPC clock-times. $R_M \geq 4$

R_R = read-register cycle time in BPC clock-times.

W_M = write-memory cycle time in BPC clock-times. $W_M \geq 4$

E = execution time of longest possible instruction.

n = number of DMA words transferred during one DMA Request.

I = levels of indirect addressing excluding the indirect in RB.

APPENDIX

M=MULTIPLICAND

X ···0 0 1 1 1 0 1 0··· (MULTIPLIER)

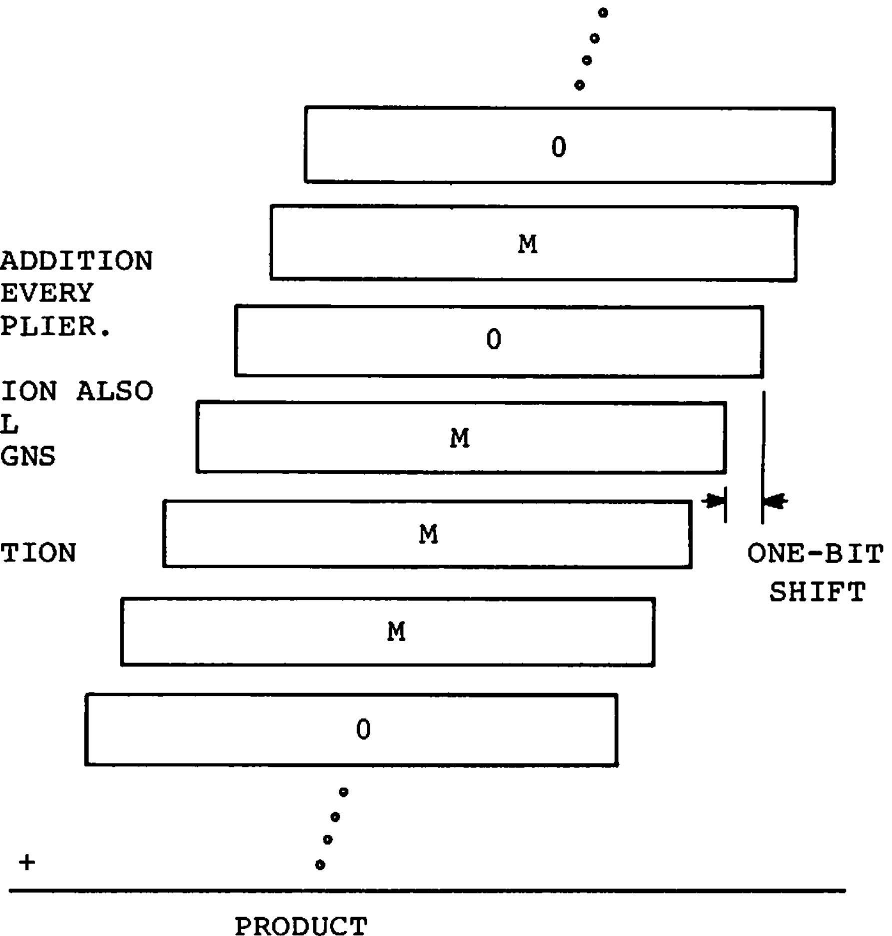
b_{i+7} b_{i+6} b_{i+5} b_{i+4} b_{i+3} b_{i+2} b_{i+1} b_i

$$\text{PRODUCT} = \sum_{i=0}^{n-1} b_i 2^i M$$

WHERE n=NUMBER OF BITS IN THE MULTIPLIER

NOTICE THAT ONE ADDITION IS REQUIRED FOR EVERY ONE IN THE MULTIPLIER.

SUCH MULTIPLICATION ALSO REQUIRES EXTERNAL INSPECTION OF SIGNS AND SUBSEQUENT COMPLIMENTING TO ALLOW MULTIPLICATION OF NUMBERS WITH DIFFERING SIGNS.



The Principle Of "Standard" Binary Multiplication.

M=MULTIPLICAND

X ···0 0 1 1 1 0 1 0···

MULTIPLIER>0

(SEE NOTES 3 & 4 CONCERNING THE SIGNS OF THE FACTORS AND THEIR PRODUCT)

DECOMPOSE THE MULTIPLIER INTO A SUM OF NUMBER COUNTING EITHER ALL ZEROS OR SINGLE SERIES OF ADJACENT ONES AND THEN DISTRIBUTE THE MULTIPLICATION.

= M X 0 0 0 0 0 0 0 0
+ M X 0 0 0 0 0 0 1 0 (E) (C)(A)
+ M X 0 0 1 1 1 0 0 0 (E)(D) (B) (A) (E)

REPLACE EACH NUMBER HAVING ONE OR MORE ONES IN IT BY ANOTHER NUMBER WITH A SINGLE ONE AND A SUBTRACTION.

= M X 0 0 0 0 0 0 0 0
+ M X 0 0 0 0 0 1 0 0 − M X 1 0 (F)
+ M X 0 1 0 0 0 0 0 0 − M X 1 0 0 0 (G)

(THE PRINCIPLE USED HERE IS THAT 1111=10000-1)

THE NUMBER OF SHIFTS ALREADY PERFORMED PRIOR TO THE TIME OF THE SUBTRACTION TAKE CARE OF MULTIPLYING M BY THESE POWERS OF TWO.

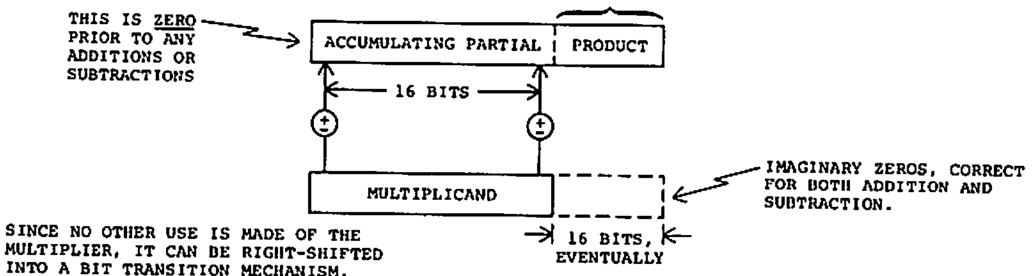
HOW THE MULTIPLIER IS USED AS IT IS SCANNED, RIGHT-TO-LEFT, ONE BIT AT A TIME:

(A) A ZERO-TO-ONE TRANSITION REQUIRES AN IMMEDIATE SUBTRACTION, FOLLOWED BY A SHIFT.

(B) SUBSEQUENT ONE-TO-ONE TRANSITIONS THEN REQUIRE ONLY WHAT WOULD NORMALLY BE REQUIRED FOR ZERO-TO-ZERO TRANSITIONS, i.e., ONE SHIFT EACH.

(C) (D) THESE ONE-TO-ZERO TRANSITIONS CORRESPOND TO ONES (F) AND (G), RESPECTIVELY, AND EACH REQUIRES AN ADDITION.

(E) A ZERO-TO-ZERO TRANSITION REQUIRES ONLY A SHIFT.



NOTES:

1. FOR PURPOSES OF DETERMINING A TRANSITION ASSOCIATED WITH THE RIGHT-MOST BIT OF THE MULTIPLIER, A ZERO IS ASSUMED TO LIE TO THE RIGHT OF THAT BIT.

2. NOTICE THAT THERE CANNOT BE A ONE-TO-ZERO TRANSITION WITHOUT PRECEEDING ZERO-TO-ONE TRANSITION. THUS, A SUBTRACTION PRECEEDS EACH ADDITION.

3. ASSUMING THE SIGN OF THE MULTIPLIER IS POSITIVE, THE SIGN OF THE PRODUCT IS THE SAME AS THE SIGN OF THE MULTIPLICAND. BUT THIS IS GUARANTEED BY THE ALGORITHM BECAUSE THE PRODUCT IS FORMED SOLELY THROUGH OPERATIONS EXACTLY EQUIVALENT TO ADDITIONS, AND BY ARITHMETIC SHIFTS. NEITHER OF THOSE CAN CREATE A RESULT HAVING A SIGN OPPOSITE THAT OF THE MULTIPLICAND.

4. MULTIPLICATION BY A NEGATIVE MULTIPLIER IS CONSIDERED IN ANOTHER DRAWING.

5. MULTIPLICATION WITH A MULTIPLICAND OF ZERO WORKS BECAUSE, NO MATTER HOW IT IS DONE, ZERO, ADDED TO OR SUBTRACTED FROM ITSELF, IS STILL ZERO.

6. MULTIPLICATION BY A MULTIPLIER OF ZERO WORKS BECAUSE THEN THERE ARE NEVER ANY TRANSITIONS TO CAUSE ANY ADDITIONS OR SUBTRACTIONS. SINCE THE PARTIAL PRODUCT STARTS OUT ZERO, IT STAYS ZERO.

Operation On Booth's Algorithm When The Multiplier Is Positive, Or When One Of The Factors Is Zero.

APPENDIX

1. IN THE EVENT THAT THE MULTIPLIER IS NEGATIVE, THE SIGN OF THE PRODUCT IS OPPOSITE THE SIGN OF MULTIPLICAND. WE SHALL DIVIDE THE POSSIBLE INSTANCES OF MULTIPLYING BY A NEGATIVE MULTIPLIER INTO THREE CATEGORIES AND SHOW THAT PROPER RESULTS ARE OBTAINED IN EACH CASE.

2. <u>CASE I</u> PRODUCT=$-1 \cdot M$

   LET M=MULTIPLICAND
   LET MULTIPLIER=-1=1111111111111111
   |← 16 BITS →|

   THIS CASE WORKS BECAUSE THERE IS AN IMMEDIATE ZERO-TO-ONE TRANSITION, CAUSING A SUBTRACTION FROM ZERO (WHICH GIVES THE PARTIAL PRODUCT A SIGN OPPOSITE THAT OF THE MULTIPLICAND).* BUT SINCE THE REST OF THE MULTIPLIER IS ALL ONES, ONLY ARITHMETIC SHIFTS FOLLOW THIS SUBTRACTION.

   THE COMPLEMENTED MULTIPLICAND IS SHIFTED TO FAR RIGHT OF THE 32-BIT ANSWER, THUS ITS MAGNITUDE (ABSOLUTE VALUE) REMAINS UNCHANGED, AND SINCE THE SHIFTS ARE ARITHMETIC SHIFTS, THE SIGN IS PRESERVED.

3. <u>CASE II</u> PRODUCT=$-2^P \cdot M$

   LET M=MULTIPLICAND
   LET MULTIPLIER=-2^P=111100 · · · 0 (P ZEROS)
   |← 16 BITS →|

   IN THIS CASE THERE ARE P LEADING ZERO-TO-ZERO TRANSITIONS, EACH OF WHICH SHIFTS A PARTIAL PRODUCT WHICH IS ZERO, AS NOTHING HAS BEEN ACCUMULATED YET. SO THOSE SHIFTS HAVE ABSOLUTELY NO EFFECT.

   THE SINGLE ZERO-TO-ONE TRANSITION CAUSES A SUBTRACTION FROM ZERO, WHICH ESTABLISHES THE SIGN OF THE PRODUCT AS OPPOSITE THAT OF THE MULTIPLICAND.* THE REMAINING ONES IN THE MULTIPLIER CAUSE 16-P ARITHMETIC SHIFTS, WHICH PRESERVE THE SIGN. BUT THESE SHIFTS FALL P SHIFTS SHORT OF FULLY SHIFTING THE COMPLEMENTED MULTIPLICAND TO THE RIGHT IN THE 32-BIT ANSWER SPACE. THIS IS AN EFFECTIVE LEFT-SHIFT OF P PLACES IN THAT 32-BIT SPACE. HENCE THE PRODUCT IS THE COMPLEMENT OF THE MULTIPLICAND, MULTIPLIED BY 2^P.

4. <u>CASE III</u> PRODUCT=$-Y \cdot M$

   LET M=MULTIPLICAND
   LET -Y REPRESENT A NEGATIVE NUMBER DIFFERENT THAN -1 OR THE NEGATIVE OF A POWER OF 2:

   $-Y \neq -1$

   $-Y \neq -2^K$

   THEN -Y CAN BE DECOMPOSED INTO THE SUM OF SOME X>0 AND -2^P FOR SOME P:

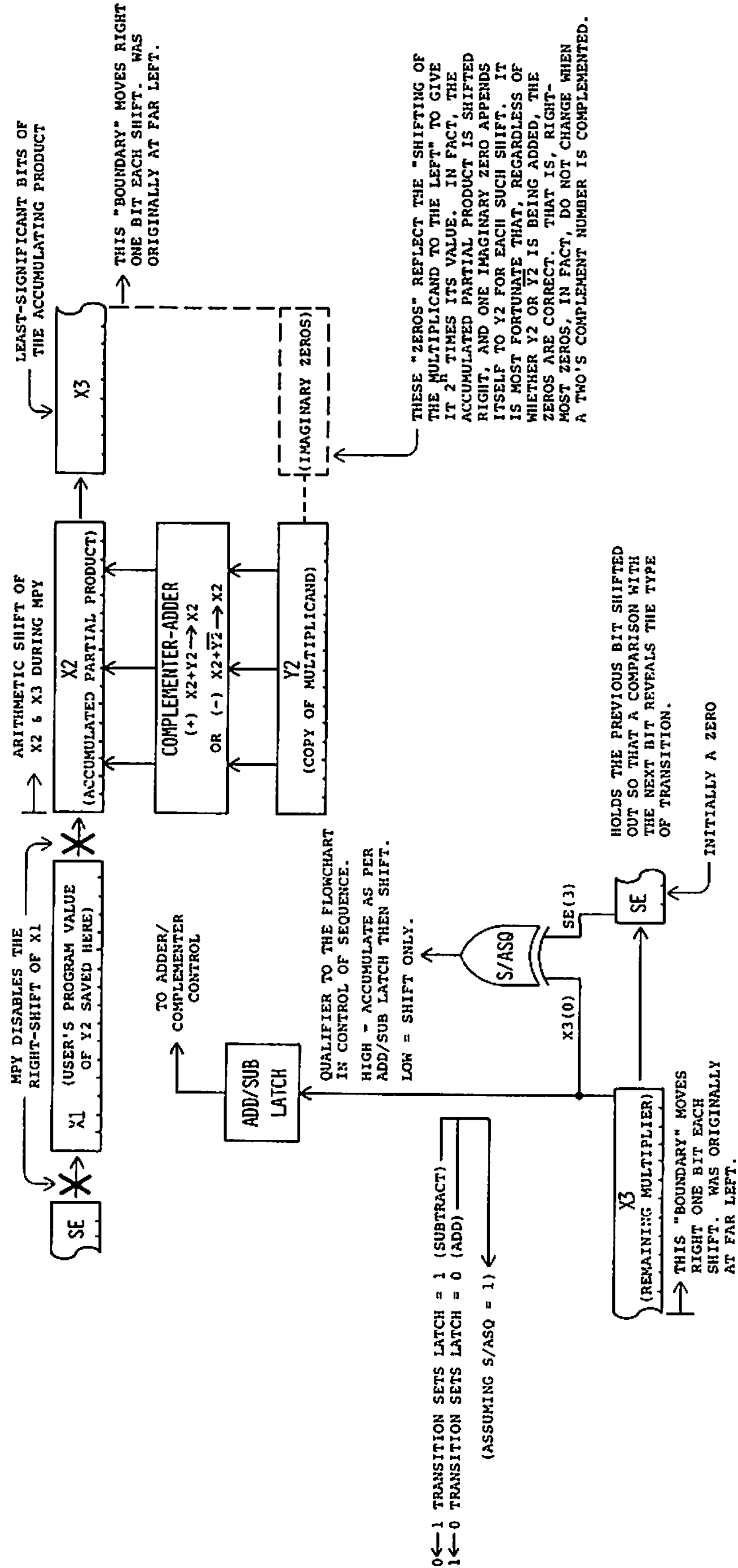
```
-Y=111010110···  =  111000000···=-2^P
                  +    010110···= X
                    ─────────────────
                    111010110···=-Y
```

   THEN, $-Y \cdot M = (-2^P + X) \cdot M = -2^P \cdot M + X \cdot M$

   AS THE MULTIPLIER IS SCANNED, X·M IS FORMED IN THE FASHION FOR POSITIVE MULTIPLIERS. THEN THE PRODUCT FOR $-2^P \cdot M$ IS ACCUMULATED TO IT. THE PROCEDURE OF THE ALGORITHM IS SUCH THAT THE FORMING OF X·M IS INDEPENDENT OF, AND DOES NOT INTERFERE WITH, THE SUBSEQUENT FORMATION OF $-2^P \cdot M$. IT IS, SO TO SPEAK, AS IF THE FORMATION OF $-2^P \cdot M$ PICKS UP WHERE FORMING X·M LEAVES OFF. THE ONLY DIFFERENCE IS THAT IN THE FORMATION OF $-2^P \cdot M$ THE MULTIPLICAND IS NOT SUBTRACTED FROM ZERO, BUT FROM X·M. THE SIGN OF THE RESULT OF THAT SUBTRACTION WILL BE OPPOSITE THE SIGN OF X·M, SINCE $2^P > X$. SINCE X·M HAS THE SIGN OF THE MULTIPLICAND, THIS MEANS THE FINAL PRODUCT HAS THE SIGN OPPOSITE THAT OF THE MULTIPLICAND, WHICH IS CORRECT.

*NOT TRUE IN 16-BIT COMPLEMENT ARITHMETIC IF THE MULTIPLICAND IS 1 000 000 000 000 000 (-32768). THE ARGORITHM FAILS WITH THAT MULTIPLICAND FOR THIS REASON. SEE THE BUG DESCRIPTION AT THE END OF THIS SECTION.

Operation Of Booth's Algorithm When The Multiplier Is Negative.



Block Diagram Of The Hardware Controlled By The Flow Chart Which Does The Booth's Multiply.

CASE I

```
1.  LET:  B = 1 000 000 000 000 000 (MULTIPLICAND=-32768)
          A = 0 000 000 000 000 001 (MULTIPLIER=1)

2.  SEQUENCE OF MULTIPLIER TRANSITIONS:
          0 000 000 000 000 001 ¦ 0<--FOR INITIAL COMPARISON PURPOSES
                                    ZERO-TO-ONE
                                    ONE-TO-ZERO
                                    14 ZERO-TO-ZERO TRANSITIONS

3.  ZERO-TO-ONE TRANSITION; SUBTRACT B FROM ACCUMULATION (i.e., 2'S COMPLEMENT
    B AND ADD IT TO ZERO-WHICH IS THE INITIAL VALUE OF THE ACCUMULATION).  THEN
    SHIFT ACCUMULATION RIGHT ONCE.
            0 111 111 111 111 111<-1'S COMPLEMENT OF B
COMPLEMENT  +                   1  ADD ONE
            1 000 000 000 000 000<-2'S COMPLEMENT OF B:
  ADD       +                   0  NOTE THAT THERE IS NO CHANGE-
            1 000 000 000 000 000  2'S COMPLEMENT OF -32768 IS -32768
  SHIFT     1 100 000 000 000 000 ¦ 0 --- --- --- --- ---

4.  ONE-TO-ZERO TRANSITION: ADD B TO ACCUMULATION, THEN SHIFT.
THIS NEVER
HAPPENS     1 100 000 000 000 000 ¦ 0 --- --- --- --- ---
  ADD     + 1 000 000 000 000 000 ¦
          1 0 100 000 000 000 000 ¦ 0 --- --- --- --- ---
  SHIFT     0 010 000 000 000 000 ¦ 0 0-- --- --- --- ---

5.  NOW THERE ARE 14 ZERO-TO-ZERO TRANSITIONS, EACH ACCOMPANIED BY A SHIFT.
                          32-BIT RESULT
AFTER 14
SHIFTS      0 000 000 000 000 000 ¦ 1 000 000 000 000 000 ¦
```

DIFFERENT RESULTS!

CASE II

```
1.  LET:  B = 0 000 000 000 000 001 (MULTIPLICAND=1)
          A = 1 000 000 000 000 000 (MULTIPIER=-32768)

2.  SEQUENCE OF MULTIPLIER TRANSITIONS:
          1 000 000 000 000 000 ¦ 0<--FOR INITIAL COMPARISON PURPOSES
                                    15 ZERO-TO-ZERO TRANSITIONS
                                    ZERO-TO-ONE

3.  FIRST THERE ARE 15 ZERO-TO-ZERO TRANSITIONS, EACH ACCOMPANIED BY A SHIFT.
    BUT THE ACCUMULATION IS ZERO TO BEGIN WITH, AND SO IT REMAINS ZERO.
AFTER 15
SHIFTS      000 000 000 000 000 ¦ 0 000 000 000 000 00-

4.  ZERO-TO-ONE TRANSITION: 2'S COMPLEMENT B AND ADD TO ACCUMULATION, THEN SHIFT.
            1 111 111 111 111 110<-1'S COMPLEMENT OF B
COMPLEMENT  +                   1  ADD ONE
            1 111 111 111 111 111<-2'S COMPLEMENT OF B
  ADD       + 0 000 000 000 000 000 ¦ 0 000 000 000 000 00-
            1 111 111 111 111 111 ¦ 0 000 000 000 000 00-
  SHIFT     1 111 111 111 111 111 ¦ 1 000 000 000 000 000
                          32-BIT RESULT

5.  THE RESULT ABOVE IS THE NEGATIVE OF THE RESULT IN THE OTHER CASE.
    MULTIPLICATION WITH -32768 IS NOT COMMUTATIVE.
                                                          1'S COMPLEMENT OF
          0 000 000 000 000 000 ¦ 0 111 111 111 111 111<- RESULT IN CASE II
        +                                             1  ADD ONE
          0 000 000 000 000 000 ¦ 1 000 000 000 000 000<-2'S COMPLEMENT OF
                                                          RESULT IN CASE II IS
                                                          SAME AS RESULT IN
                                                          CASE I
```

Bug In MPY

APPENDIX

CHIP	DESCRIPTION	VERSION 15-bit	VERSION 16-bit
BPC	- EXE of, or instruction fetch from, an addressable register in the BPC fails to give SMC	√	
IOC	- DDR not reliable	√	
	- IOC releases $\overline{\text{INT}}$ at wrong time to allow single level indirect for interrupt vector	√	
	- IOC doesn't allow IOC machine-instructions to be fetched from its own registers	√	√
	- Glitch on $\overline{\text{BYTE}}$	√	√
	- Pulse Count Mode unuseable due to "timing difficulties"	√	---
EMC	- Multiplication with -32768 is not commutative	√	√
	- CMX not useable with DMA	√	
ALL	- POP synchronizer is unreliable?	---	√

Processor Bug List

Also:

Currently, during a DMA write-into-memory operation, the IOC gives Buffer Enable ($\overline{\text{BE}}$) a half-state too soon. This (wrongfully) allows both the peripheral and the IOC to drive the IDA Bus for a short time. The problem is shown in IOC figure 24-17, note I, in the *How They Do Dat Manual*. The statement there that this causes no problems is false. It goofs up the AEC (Address Extension Chip) if the relative speeds of the various chips in the system is just right. Accordingly, at this time (June '78) some sort of fix to the IOC is being contemplated.