MIT INSTRUMENTATION LABORATORY

APOLLO

GUIDANCE, NAVIGATION AND CONTROL

Approved: _____ Date: Feb 67
E. M. COPPS, JR., DIRECTOR, GUIDANCE PROGRAMING, APOLLO GUIDANCE AND NAVIGATION PROGRAM

Approved: _____ Date: 20 Feb 67
DAVID G. HOAG, DIRECTOR
APOLLO GUIDANCE AND NAVIGATION PROGRAM

Approved: _____ Date: 20 Feb 67
RALPH R. RAGAN, DEPUTY DIRECTOR
INSTRUMENTATION LABORATORY

E-2052

Vol I of II

AGC4 BASIC TRAINING MANUAL

by
Bernard I. Savage*
Alice Drake*

January 1967

INSTRUMENTATION LABORATORY
CAMBRIDGE 39, MASSACHUSETTS

COPY # _____

ACKNOWLEDGMENT

Many people have helped us while we were writing this manual. We are particularly grateful to Hugh Blair-Smith for reviewing the manual and for his suggestions, and to Charles Muntz for his expertise and consultation on many of the systems programs.

This report was prepared under MIT Instrumentation Laboratory Purchase Order No. ILK 222242 and under DSR Project 55-23850 sponsored by the Manned Spacecraft Center of the National Aeronautics and Space Administration through Contract NAS9-4065.

The publication of this report does not constitute approval by the National Aeronautics and Space Administration of the findings or the conclusions contained therein. It is published only for the exchange and stimulation of ideas.

E-2052

AGC4 BASIC TRAINING MANUAL

ABSTRACT

This manual contains **a** concise description of that which a computer programmer should know about the Apollo Guidance and Navigation Programming System to be useful. That is we answer the following questions: What are the Pertinent machine characteristics? What programming languages and conventions exist for my use? What systems subroutines may I rely upon? How do I communicate with the system subroutines which I need? This manual does not concern itself with the Mission Programming System or that which an engineer or mathematician must know to adequately program a phase of the mission after he has an adequate knowledge of the system.

This manual attempts to be thorough while brief. It does not try to exhaust all there is to know about a subject nor does it try to make the reader an expert on any subject. It is designed *so* that someone fairly new to the subject may acquire a practical understanding of it within the shortest time. Whenever a detailed and complete understanding is required the reader should consult the program listing and/or other technical documents.

This manual is divided into four sections. Section I discusses the AGC4 and how to program it in Assembly Language, Section II describes the Interpreter and how to program in Interpretive Language. Section III describes the System Software subroutines and how to interact with them. Section IV contains an outline and suggestions for teaching sections I-III. Each section has a table of contents.

by A. L. Drake
B. I. Savage
Computer Consultants, Incorporated
January 1967

TABLE OF CONTENTS

I. BASIC TRAINING DOCUMENT: BASICS OF AGC PROGRAMMING

1.1 Introduction and Memory Outline

A word in AGC memory consists of 15 binary bits, schematically numbered from left to right as bit 15, 14, . , ., 1. A sixteenth bit called the parity bit is inaccessible to the programmer but serves as a check against hardware malfunction. When a word is stored in memory, the count of the number of bits in the word which are set to 1 must be odd. If the count equals an even number, the parity bit will be set to 1 *so* that the count is odd; otherwise the parity bit is set to 0. When the same word is read from memory, the hardware ascertains that an odd number of bits came from memory. If not, the implication is that **a** bit was lost. This is called a parity error and results in special processing.

15 14 13 . . . 1 P

Each word in AGC memory may be interpreted as data or as an instruction.

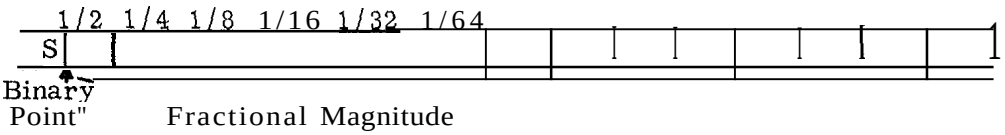
1.1.1 ,Data Representation

One word by itself constitutes a Single Precision (SP) quantity. Bit 15 is the sign and bits 14—1 have magnitude of $2^{14} - 1$. If bit 15 = 1, the magnitude is negative and is represented as the ones complement of the positive magnitude (discussed below). Bit 14 is the high order bit (highest value) and bit one is the low order bit (lowest value).

$\pm . . 2^{14} - 1$ negative ones (1's) complement

15, 14, . . ., 1

For arithmetic purposes the value in bits 14— 1 is thought of as a fraction. That is, the binary point is between the sign and bit 14. For instance, a one in bit 14 is equivalent to 1/2. From a programmer's point of view, the programmer must keep track of the "imaginary" point's position within the word in. accordance with the appropriate scaling.



Fourteen magnitude bits may not always allow us sufficient precision. Thus we may represent data in a Double Precision (DP) quantity within two adjacent words of memory. Since each single precision word has 14 magnitude bits, the combined quantity has 28 bits with a precision of 2^{28}-1. Bit 15 of the first word contains the sign. Bit 15 of the second word will normally be the same as bit 15 of the first word but may differ in certain cases. Bits 14-1 of word 1 represent the high-order bits and bits 14-1 of the second word represent the low-order bits. All 28 bits exist in complemented form if the sign (s) is negative.

	Word 1		Word 2
+	High	X	Low
-			
15	14. 1	15	14. 1
	28 15		14

For even greater accuracy, a quantity may be contained within 3 adjacent words and is called a Triple Precision (TP) quantity. (The third word serves the same function in TP as word 2 does in DP.) In essence, we add 14 low order bits so that we may represent a value of 2^{42} - 1 (thought of as a fraction, we would say $1 - 2^{-42}$.) Again, negative value would be represented in one's complement form within all 3 words,

	Word 1		Word 2		Word 3
+	High	X	Middle	X	Low
-					
15	14. . . 1	15	14. 1	15	14. . . 1
	42 29		28 15		14 1

Three double precision quantities are used to represent a 3-dimensional vector, Each DP "word" contains the value of one component of the vector. Again, all 6 words must be adjacent and, normally, the first two words represent the X component, the next two words represent the Y component, and the last two words represent the Z component, Of course, the sign of each DP component need not be the same.

+	Word 1	Word 2	+	Word 3	Word 4	+	Word 5	Word 6
-			-			-		
15	high $_X$-	low	15	high $_y$-	low	15	high $_Z$	low

Lastly, an AGC word may be thought of as a full 15-bit quantity, where all 15 bits are magnitude without sign, representing $2^{15} - 1$. This representation could be used to make a word into a counter. For logical purposes, each bit or some combination of bits may be used as an indicator or may serve boolean purposes.

The access time for taking one word from memory is approximately 12 microseconds or one memory cycle time (MCT).

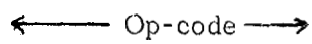
1.1.2 Instruction Representation

The 15 bits of an AGC word may be selected and executed by the AGC as an instruction. In this case, 3 bits, 15 – 13, form an octal value from 0 – 7 and represent the op-code. The encoding of the op-code is what determines the particular behavior of each instruction, As 3 bits have been specified for op-code selection, we may have $2^3 = 8$ basic machine instructions, and indeed we do. Thus, any word taken by itself forms a legal instruction. This implies that a data word may be executed yielding storage and unexpected results, and a programmer must take pains to keep his data (constants, for instance) separate from his instructions. Actually, we shall later encounter a way of extending the basic machine instructions (discussed below) by using certain 2-word instruction sequences or by extending the 3-bit op-code to include bits 12 and 11 for op-code purposes for instructions which apply to erasable memory.

The remaining 12 bits of an instruction word form the address portion. (Depending upon the op-code, the address is used to render accessible the contents of the specified memory location or is used as a number to point to a location in memory (to transfer control to a location, for example.) Twelve bits may form an address for the range 0 – 7777 octal or 2^{12} (4096) decimal locations. The system requirements necessitate a much larger memory store. Thus, the address portion is encoded *so* as to be combined with another location called the Bank Register, allowing us *to* form an effective address of 15 bits for the range $0 - 2^{15} - 1$. Further encoding allows combination with an indicator called the Super Rank Bit which enables us to form a 16-bit effective operand address for the range $0 - 2^{16} - 1$. This method of encoding the address portion of an instruction word is discussed under Addressing, below.

Op - Code	Address
15 14 13	12 1
0 – 7	0 – 7777
8 cases	4096 decimal addresses

1.1.3 Quarter Codes



15 14 13 12 11 10 1

In cases where only the 10 low-order bits are necessary to form the address portion of a word, the 3-bit op-code may be extended by use of bits 12 and 11 to form a 5-bit op-code nicknamed a quarter code. Quarter codes allow us to use more than 8 op-codes when addressing erasable memory (as discussed below).

1.1.4 Layout Memory

The AGC memory may be divided broadly into erasable and fixed memory. Erasable memory is in the range 0 – 3777 octal. The contents of a location in erasable memory may be altered by writing into it. Fixed memory is in the range 10000 – 117777 octal, but there are gaps. That is, not every location corresponding to one of the addresses in this range exists. Fixed memory is a "read only" memory. The programmer may not alter the contents of any location in fixed memory, and in fact the hardware will not permit it. Thus, only 2048 decimal locations exist for ordinary programming requirements that need modifiable storage. This is very little memory, and one crucial requirement of programmers is to design and implement programs which use a minimum amount of erasable storage (the use of temporaries and switches is one example.)

1.1.5 Erasable Memory

Although erasable memory is defined as that portion of memory locations within the range 0 – 3777 octal, an erasable memory location must also meet the requirement of being defined within the 10 low-order address bits, because bits 12 and 11 must = 0 as a "signal" to the hardware that E-memory is being addressed, While no problem arises in addressing locations 0 – 1377 octal, the use of an 11th bit is sometimes necessary in addressing the range 1400 – 3777 octal. For addresses within this range, then, we use a 3-bit Erasable Bank Register in conjunction with the 8 low order address bits to form an effective address.

1.1.6 Special Registers

The first 60 locations of erasable memory are used exclusively as special registers. The accumulator (A)—octal pseudo-address 0000—is a 16-bit arithmetic element. Bit 15 contains the sign, which is duplicated into bit 16. Bits 14 . . .1 contain the magnitude of the quantity. Bit 16 is used to indicate the corrected sign in the case of overflow (discussed below),

The lower product register (L)—octal pseudo-address 0001— is a 15-bit register which forms the lower part of the accumulator when Double Precision quantities are used. It contains the 14 least significant bits of a product after multiplication and the remainder of a quotient after division.

The *Z* Register— octal pseudo-address 0005— serves as a 12-bit program counter. It contains the next address in memory from which an instruction will be fetched. These 12 bits are inadequate to address all of memory and may be combined with bank bits to form up to a 16-bit address. This shall be discussed under Addressing. The 16-bit Q Register contains, after a Transfer Control (TC) instruction, what would normally be the contents of the *Z* Register. For example, when a "TC" instruction is executed at location L, the contents of the *Z* Register contain the address of the instruction to which the program has transferred control. The Q Register contains the address of the instruction following the "TC" instruction, or L+1. If the instruction had been any other than a "TC", this address would have been contained in the *Z* Register. When the subroutine initiated by the "TC" instruction is finished, a "TC to *Q*" instruction will return control to the instruction following the "TC" instruction in the main program, or L + 1.

The *Zero* Register— octal pseudo-address 0007— always contains only zeroes. When referenced, it will yield zeroes. One use of this is as a constant to clear a desired location.

A value may be altered by writing it into one of 4 special registers. When a quantity is written into the Cycle Right Register— octal pseudo-address 0020— bit 1 goes into the sign bit, while bits 15 . . , 2 shift right 1 bit. When a quantity is written into the Shift Right Register— octal pseudo address 0021—the sign bit is duplicated into bit 14, while bits 14 , . . 1 are shifted right 1 bit. By faithfully reproducing the sign bit, we preserve the algebraic integrity of the value. The original contents of bit 1 are lost. Shifting right n places is, of course, the equivalent of dividing by 2^n. When the Cycle Left Register - octal pseudo-address 0022 – is written

into, the sign bit goes into bit 1, while the contents of bits 14 . . . 1 shift left 1 bit. When the Edit Op-Code Register - octal pseudo address 0023 - is written into, the sign bit is lost, while bits 14 - 1 are shifted right 7 places, displacing the original contents of bits 7 . . . 1, which are consequently lost. This last register is not of general interest. It is used in implementing interpretive instructions.

Editing Register Transformations

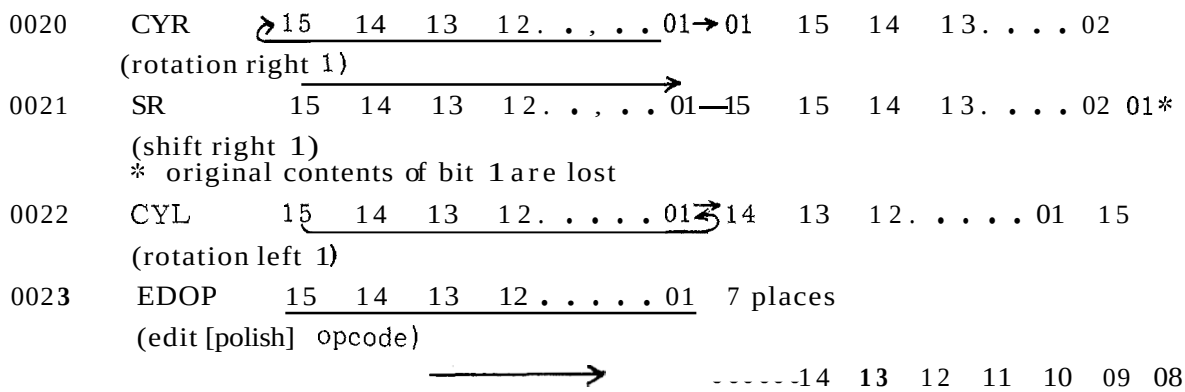


Figure 1

1.1.7 Fixed Memory

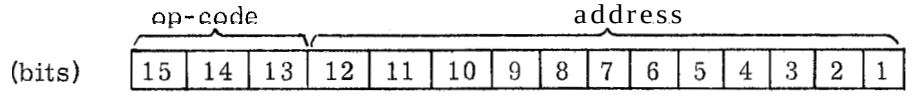
Fixed memory is that portion of memory addresses in the range 4000 – 117777 octal. These memory addresses are divided into 36 banks of 1024 words each. The first 2 banks of fixed memory – banks 02 and *03* – with addresses 4000 – 7777, are known as "Fixed-Fixed" memory. Notice that Fixed-Fixed memory can be defined within the 12 address bits.

The remaining 34 banks of fixed memory, with addresses in the range $10000-117777_8$, need additional bits within which to fully define their addresses. For these cases, a 5-bit Fixed Bank Register, which is more definitively discussed below, is made available for combination with the 10 low-order address bits. These 34 banks, which require the use of a bank register, are known as "Fixed-Switchable" memory. Addresses in the range $100000-117777_8$ require 16 bits for definition. A 16th bit is provided for combination with the FCADR (Fixed Bank Complete Address), i. e. with the 5 bits from the FB and the 10 low-order address bits. Addresses in the range 110000 – 117777 comprise Super Bank 4*. Addresses in the range 70000 – 107777 comprise Super Bank 3. The reason for this will be fully discussed under Addressing.

* Super Banks *0* and 1 have been renamed 3 and 4, respectively.

1.2 Addressing

Within the 15-bit word in AGC memory, we have only a 12-bit address field to reference the 38,912 decimal locations in memory.

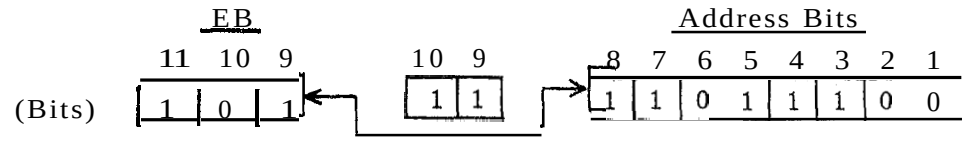


Since many memory locations require a 13- to 16-bit address field for definition, the following addressing schemes have been developed:

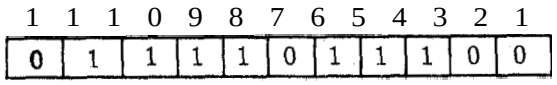
(It is first important to distinguish between the terms "address", "pseudo-address", and "effective operand address". "Address" refers simply to the 12-bit address portion within a 15-bit word in memory. The "pseudo-address" (PA) is the absolute address of memory locations $0-117777_8$. The term "pseudo-address", or "absolute address", is used when discussing fixed-switchable memory addressing where the absolute addresses are always 10000_8 more than their machine representations. The "effective operand address" (EOA) is the final address formed by the hardware at the execution time.)

The hardware recognizes a 00 configuration in bits 12 and 11 as a "signal" that the address refers to erasable memory, which we have said must be defined within the 10 low-order address bits of a word. If bits 12 and 11 are equal to 00, the hardware tests bits 10 and 9. An 11_2 configuration in bits 10 and 9 indicates that the pseudo-address is in Erasable-Switchable memory (i.e. in the range $1400-3777_8$) and that we need the use of the 3-bit Erasable Bank Register (EB). The combination of the 3 bits from the EB and the 8 low-order address bits provides an 11-bit address field, which is sufficient for the definition of all Erasable-Switchable absolute-addresses.

We set the EB equal to the particular bank number in the range $0-7$, which would be defined ordinarily in bits 11, 10, and 9 if we had the use of the 11 low-order address bits for defining Erasable-Switchable addresses. The configuration in the 8 low-order address hits is the $0-377_8$ ($=256_{10}$ bank locations) augment within the bank specified in EB. For example, since the absolute address 2734_8, which looks like 10 111 011 100_2 in machine representation, requires more than 10 address bits for definition, we set EB equal to the value specified in bits 11, 10, and 9, or $101_2 = 5_8$. We also set bits 10 and 9 equal to 11_2 *so* that at execution time the hardware will fetch the 3 EB bits, drop bits 10 and 9 (which is the equivalent of subtracting 1400 from the 10-bit address), and append the EB bits to the 8 low-order address bits. We now have

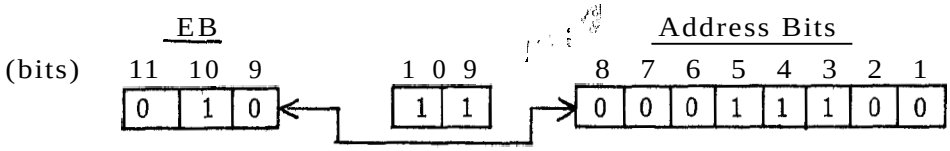


The absolute address 2734_8, then, can be expressed as an augment of 334_8 in EB 5. We combine the above to get:

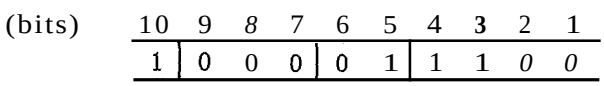


or 1734_8 from which 1400 is subtracted.

Any configuration in bits 10 and 9 of the address other than 11_2 indicates to the hardware that the pseudo-address is within erasable memory below 1400_8 and can be defined within the 10 low-order bits of a word. In this case, there is obviously no need for the use of a bank register. For example, the address 1034_8 would look like $1\,000\ 011\ 100_2$ to the hardware. Finding no 11_2 configuration in bits 10 and 9, the hardware would merely form a 10-bit address field. Of course, it is possible to address all of erasable memory via the EB. For instance, the absolute address $1034_8 = 001\ 000\ 011\ 100_2$, which we considered above, can be handled thus. We set the EB to $010_2 = 2_8$ and bits 10 and 9 equal to 11_2. At execution time, the hardware, sensing the 11_2 configuration in bits 10 and 9, fetches the EB bits, drops bits 10 and 9, and appends the EB bits to the 8 low-order bits, giving us



which is the expression of the absolute address 1034_8 as an augment of **34** within EB 2. This is obviously the equivalent of the configuration of 1034_8 as a low-order address

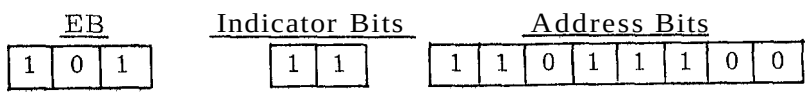


While it is therefore possible to address erasable memory below 1400_8 via the EB, it is usually preferable to define these addresses within the 10 low-order address bits of a word.

A step by step recapitulation of addressing Erasable-Switchable absolute addresses follows:

1. $2734_8 \rightarrow 10\ 111\ 011\ 100_2$.
   The octal address is converted to machine language.
2. The programmer sets the EB to 0—7 (in this case, 5)—the value specified in bits 11, 10, and 9 ($10\ 111\ 011\ 100_2$).

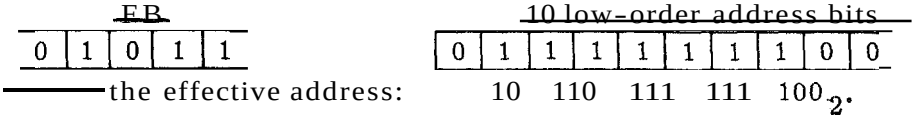
3. The programmer sets bits 10 and 9 of the address equal to 11_2 to indicate to the hardware the need for the EB (the equivalent of adding 1400_8).
4. At execution time, the hardware fetches the EB bits (101), drops bits 10 and 9 (= subtracting 1400_8 from the 10-bit address), and appends the 3 bank bits to the 8 low-order address bits. We now have



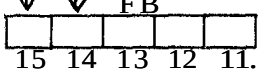
5. The above is combined to render the EOA 10 111 011 100_2.

We have now to consider the addressing schemes which develop when the hardware tests bits 12 and 11 and finds a configuration other than 00. A one in bit 12 indicates that the address is in Fixed-Fixed memory (i.*e*. in the range $4000-7777_8$), which we have defined as those addresses which require for definition no more than the 12 bits of the address field of a word, For example, the address 5467_8 which is equivalent to 101 100 110 111_2 and has a one in bit 12, can indeed be defined within the 12 address bits and is indeed within the range $4000-7777_8$. The address 7601_8, which is 111 110 *000* 001_2 in machine language, can likewise be defined within the 12 address bits and is within the range $4000-7777_8$.

On the other hand, if an address cannot be defined within a 12-bit address field, a 01_2 configuration in bits 12 and 11 indicates that the address is in Fixed-Switchable memory (i.e., in the range $10000-117777_8$) and will therefore require the use of a 5-bit Fixed Bank Register (FB). In Fixed-Switchable memory, an address is always 10000_8 more than its actual address representation in the machine since Fixed-Switchable memory begins in FB 0. Therefore, the first consideration to a programmer in converting the pseudo-address to a representation that will permit the AGC to form an effective operand address is to subtract 10000 from the address. The programmer then sets the FB equal to the value in the range $0-37_8$ which would ordinarily be specified in bits 15, 14, 13, 12 and 11 if we had the use of a 15-bit address field. Let us take as an example the address 36774_8 which becomes 26774_8 after subtracting from it 10000_8. Since its machine representation is 10 110 111 111 100_2, the programmer sets the FB equal to 13_8 (010 110 111 111 100_2), sets (usually via the Assembly) bits 12 and 11 equal to 01_2, and leaves bits $10-1$ unaltered. At execution time, the hardware senses the 01 configuration in bits 12 and 11, fetches the 5 bits *from* the FB, and masks out all but the 10 low-order address bits of the word. Masking out bits 12 and 11 is obviously the equivalent of subtracting 2000_8 from the 12-bit address. The 5 bits from the FB are now appended to the 10 low-order address bits, giving us the address 10 110 111 111 100_2, identical to the original address minus 10000_8 (26774_8).



After we have formed the EOA, the hardware tests bits 15 and 14 of the FB:



An 11_2 configuration in bits 15 and 14 indicates that a 16th bit may be required for address definition. If bits 15 and 14 are not equal to 11_2, the 15-bit address field provided by the combination of the FB with the 10 low-order bits will be sufficient for defining the effective operand address. In this case, we would follow the procedure outlined above. A step by step description of the changes which affect the address 36774_8 follows:

1. 10000_8 is subtracted from the pseudo-address 36774_8 giving us the effective operand address 26774_8.

2. $26774_8 \rightarrow 10\ 110\ 111\ 111\ 100_2$
The octal address is converted to machine language.

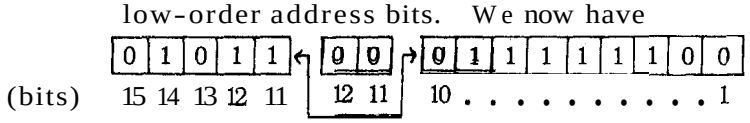
3. The programmer must provide that at execution time, the FB is set to 13_8 — the value specified in bits 15, 14, 13, 12, and 11.
($010\ 110\ 111\ 111\ 110_2$)

4. Now we mask out all but the 10 low-order address bits of the word, giving us — — — — — 0 111 111 100$_2$
(bit position) 15 . . . . . 11 10 1

5. The programmer (via assembly) sets bits 12 and 11 equal to 01_2, thereby indicating the need for the FB and now giving us — — — 010 111 111 100$_2$
15 12 1 (bit positions)

The value of the 12-bit address field for all Fixed-Switchable addresses is in the range $2000-3777_8$, which we obtain by always adding 2000 (i.e. bits 12 and 11 = 01) to the 10-bit augment of 0 — 1777 (i.e. $0-1023_{10}$ bank size).

6. At execution time, the hardware fetches the 5 bits from the FB; drops bit 11, which is the equivalent of subtracting 2000_8 from the 12-bit address; and appends the FB to the 10 low-order address bits. We now have



7. Since the address does not require a 16th bit for definition (bits 15 and 14 ≠ 11_2), the only step remaining is to combine the above to render the effective operand address
$10\ 110\ 111\ 111\ 100_2 = 26774_8$.

8. $26774_8 + 10000_8 = 36774_8$. The effective operand address is the Machine Equivalent of the pseudo-address.

All the Fixed-Switchable addresses in the range $10000-107777_8$, whose effective operand addresses are between 0 and 77777_8, can now be referenced within the 15-bit address field provided by the combination of the 5-bit Fixed Bank Register and the 10 low-order address bits of a word, There are Fixed-Switchable addresses, however, in the range 110000 - 117777, whose effective operand addresses (between 100000 and 107777_8) require a 16th bit for definition. We therefore provide a 16th bit called a Super Bank Bit or Fixed Extension Bit (FEB) in the following fashion. The hardware recognizes an 11_2 configuration in bits 15 and 14 of the FB -

FB

15	14	13	12	11
1	1			

as a signal to fetch a 16th bit - the Super Bank Bit - and append it to the Fixed Complete Address (FCADR).

For those Fixed-Switchable addresses in the range $70000-107777_8$, with effective operand addresses in the range $60000-77777_8$, the Super Bank Bit must contain a *0* (since the addresses can be defined within a 15-bit address field). For the range 110000 - 117777 (EOA $100000-107777_8$) the Super Bank Bit must = 1. Let us consider for example the address 76453_8, which becomes $66453_8 = 110\ 110\ 100\ 101\ 011_2$ in machine representation. The programmer must provide that at execution time, the FB is set to 33_8 (11011_2) - the value specified in bits 15, 14, 13, 12, and 11 ($110\ 110\ 100\ 101\ 011_2$) (bits 15 and 14 of the FB are equal *to* 11_2) - and that the Super Bank Bit is equal to *0*. At execution time, the hardware tests bits 15 and 14, and, sensing an 11_2 configuration in bits 15 and 14, it fetches and interrogates the Super Bank Bit.

When the hardware senses a 0 in the Super Bank Bit, it merely forms a 15-bit address field just as for Fixed-Switchable addresses below FB 30. For illustrative purposes (i.e. this is not a description of how the hardware actually works), the value found in bit 16 is added to the value in bit 14. If, as in this case, the Super Bank Bit is equal to 0, the configuration in bit 14 will not be altered by adding 0 to it, and there will consequently be no overflow out of bit 15.

Super Bank Bit

+0

110 110 100 101 011

110 110 100 101 011

15 1

Now, consider that the pseudo-address range $70000-77777_8$ (within FB 30–33) and the pseudo-address range $110000-117777_8$ differ by 20000_8 (*or* a 1 in Bit 14), though we also define FB 30 – 33 for the range $110000-117777_8$. We distinguish them thus: the address range $70000-77777_8$ within FB 30–33 has an 011 configuration in the Super Bank Bit while the address range $110000-117777_8$ within FB 30 – 33 has a 100 configuration in the Super Bank Bit (bit 16).

We have just discussed the disassembling of address 76453_8. Let us now consider the address 116453_8. Subtracting 10000_8, we obtain the effective operand address $106453_8 = 1\ 000\ 110\ 100\ 101\ 011_2$. The programmer makes certain that, at execution time, all but the low-order 10 address bits are masked, and that the FB and Super **Bank** Bit are set. Since the hardware interrogates the Super Bank Bit only on sensing a 11_2 configuration in bits **15** and 14, and since bits 15 and 14 are *00* in the pseudo-address -106453_8, i. e.

1	000	110	100	101	01	2
(bits) 16	15	14			1	

the programmer has had to set the FB to 33_8 (11011_2), *so* that 15 and $14 = 11_2$.

This is the same configuration as that of the FB for the effective operand address (EOA) 664538 above. Again, to distinguish between the addresses, we can think of adding the value of the Super Bank Bit to the value of bit 14. Thus,

FEB	FB				
1	1	1	0	1	1
16	15	14	13	12	11

10 low-order address bits										
0	1	0	0	1	0	1	0	1	1	
10	9	8	7	6	5	4	3	2	1	

which gives us

FEB	FB				
1	0	0	0	1	1
16	15	14	13	12	11

10 lower address bits										
0	1	0	0	1	0	1	0	1	1	
10	9	8	7	6	5	4	3	2	1	

We now have the exact binary configuration of our original EOA 106453_8 and have succeeded in differentiating between FB 33_8 for 106453_8 and FB 33_8 for the EOA 66453_8.

1.2.1 Bank Summary

The Erasable Bank Register, as we have such, is a 3-bit register which, by having its 3 bits specifying a bank number in the range $0-7_8$ appended to the 8 low-order address bits of a word, will provide us with an 11-bit address field. Within an 11-bit address field, we can define all of erasable memory (i.e. addresses $0-3777_8$).

By writing a bank number in the range $0-37$ into the 5-bit Fixed Bank Register and having the 5 bits of the FB appended to the 10 low-order address bits of a word, we obtain a 15-bit address field. This 15-bit address field is sufficient for defining all of fixed memory except those pseudo-addresses in the range $110000-117777_8$ (i.e. 15 bits is sufficient for defining addresses in the range $10,0000-107777$, after having subtracted 10000_8).

The Super Bank Bit, which is not a bank or register, provides us with the 16th bit necessary for defining the absolute addresses in the range $110000-117777_8$. Thus is all memory made addressable.

The EB, FB, and BB are all locations in memory. A bank number written into EB or FB is automatically written into BB, and information written into BB is automatically written into EB and FB.

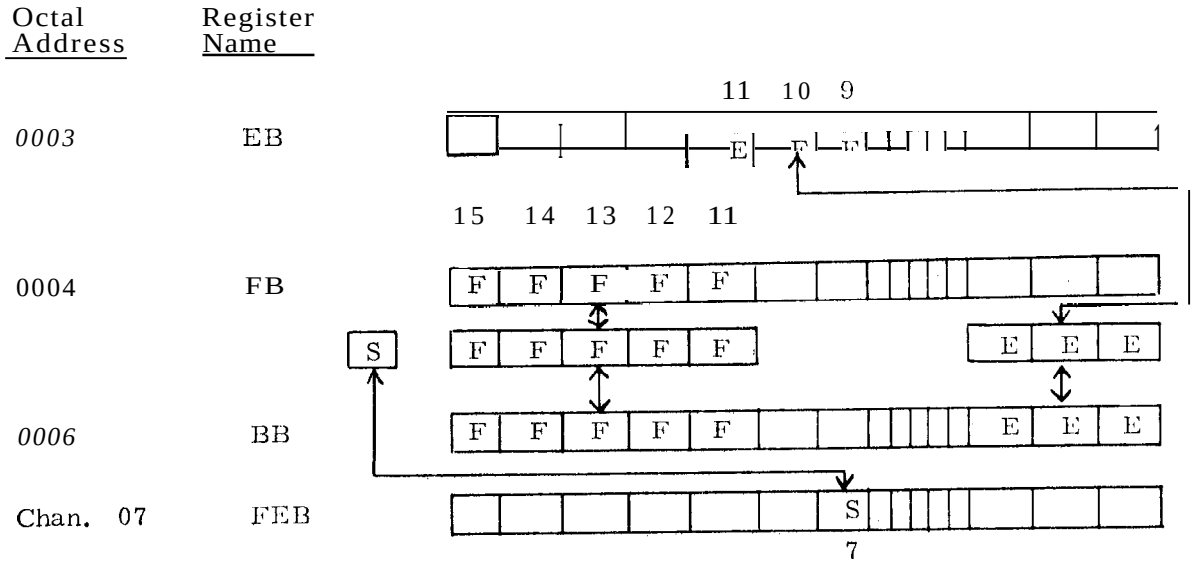


Figure 2

The programmer sets bank registers by creating constants (via assembly process) which are written into the bank registers by his program at execution time.

A special 12-bit hardware register exists called the S Register, which is inaccessible to the programmer and contains the 12-bit address portion of the referenced word of memory. Depending upon the configuration in bits 12 and 11 and in bits 10 and 9 of the contents of the S Register, the hardware will form an 11-bit address, or a 16-bit address. These effective operand addresses then go to the address selection logic for selecting the referenced address. A diagram of the logic upon which the hardware will form an 11-, 15- or 16-bit EOA from the 12-bit address in the S Register is presented in an appendix at the end of this document, (pages 1-67, 1-68).

As we have discussed previously under Special Registers, the *Z* Register has only 12 bits within which to reference all $38,912_{10}$ memory locations. In order to address up to a 16-bit absolute address, the *Z* Register bits are combined with EB or FB and FEB bits, as in the previously discussed procedure, to obtain a fixed absolute address.

Changing banks requires that the programmer has set the EB or FB to the proper configuration of the bank he wishes to go to, and that he has set the *Z* Register so that the hardware will form the proper EOA (with the aid of the bank register, if necessary), Similarly, when we wish to fetch data from one bank while we are in another bank, we must set the EB or the FB properly for combination with the address field of the fetch. One problem with fetching information from one fixed bank while we are in another is that we may lose control from the bank we are in to the bank containing the desired data to which the *Z* Register will be pointing. For example, let us consider that we are in FB 23 and that we wish to fetch data from FB 20. By executing any instruction which will fetch the data from FB 20, the *Z* Register will be set so that when the bank bits are appended, we shall then be in FB 20 rather than in FB 23 where we wish to be. Although methods of evading this problem will be discussed below, one common solution is to fetch the desired data via erasable memory, in which the previous setting of the FB will not be altered.

At the end of this document in an appendix is a diagram showing how the 38,912 memory locations fit the addressing schemes previously discussed (page 1-66).

1.3 Instruction Representation

As previously stated under Instruction Representation, the 15 bits of an AGC word may be selected and executed as in instruction. Since only bits 15, 14, and 13 of the instruction are specified to represent the op-code, we have only 8 op-codes with which to work.

We therefore introduce a 16th bit called an extracode bit, which, when appended to the 3-bit op-code and set to 1, provides us with twice the number of instructions, giving us 16 op-codes. The extracode bit is set by an "Extend" instruction and is reset by any instruction other than an "index" instruction.

Also, we have stated under Addressing that a 00 configuration in bits 12 and 11 indicates that we are referencing erasable memory, If we are able, then, to detect by the very nature of the instruction that we are addressing only erasable memory, we may use bits 12 and 11 to represent op-codes. We call the combination of the 3 op-code bits and bits 12 and 11 (when an instruction refers only to erasable memory) a quarter code (QC). The combination of the extracode bit, the 3 op-code bits, and bits 12 and 11 gives us a maximum of 6 bits for representing op-codes, thus giving us $2^6 = 64$ possible op-codes in the range $0 - 77_8$. In reality, however, there are less than forty instructions.

The following diagram presents the code names and high-order bit configurations of the 15 non-extracode instructions and the 19 extracode instructions. A detailed explanation of each instruction will he given in Section n.

1.3.1 Arithmetic and Overflow

This brief discussion of the AGC mechanization of the arithmetic unit is given from strictly a programming point of view and is therefore intended only to give some basis for analyzing program performance.

Data is represented in the *RGC* with the positive (0) or negative (1) sign of the magnitude in bit 15 and the binary magnitude within the range $0 - 2^{14} - 1$ in bits 14—1. Positive data is represented within bits 14—1as binary magnitude up to $2^{14}-1$.

bits 15-13=	0	1	2	3	4	5	6	7
Non-extracode instructions	**RELINT** INHINT EXTEND TC (TCR)	**CCS** bits 12, 11 ***= 00*** **TCF** bits 12, 11 ≠ 00	DAS bits 12, 11 ***= 50*** **LXCH** bits 12, 11 = 01 **INCR** bits 12, 11 = 10 ADS bits 12, 11 = 11	CA (CAF) (CAE)	**CS**	Resume (17) Index (NDX) bits 12, 11 = 00 **DXCH** bits 12, 11 = 01 TS bits 12, 11 = 10 XCH bits 12, 11 = 11	AD	MASK (MSK)
Extracode instructions	READ WRITE RAND WAND 30R WOR RXOR . . .	DV (bits 12, 11 = 00) BZF ≠0	MSU bits 12, 11 = 00 QXCH bits 12, 11 = 01 AUG bits 12, 11 = 10 DIM bits 12, 11 = 11	DCA	DCS	INDEX (NDX)	SU bits 12, 11 = 00 BZMF	MP
bits 16-13=	10	11	12	13	14	15	16	17

Figure 3

For example, the positive octal quantity 7305_8 and its negative one's complement would be represented in an *RGC* word as :

positive	0	0	0	1	1	1	0	1	1	0	0	0	1	0	1
bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
negative	1	1	1	0	0	0	1	0	0	1	1	1	0	1	0

sign bit

The sum of the negative and positive representation of a quantity will obviously equal a configuration of all one's. Consider the sum of the two previous examples:

+7305 =	0	0	1	1	1	0	1	1	0	0	0	1	0	1
+ -7305 =	1	1	0	0	0	1	0	0	1	1	1	0	1	0
-0000 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1

or

$-0 = 77777_8$

Because the AGC uses only one's complement arithmetic when under program control, the quantity "zero" has two possible representations: 00000_8 and 77777_8, which are designated respectively as *+0* and *-0*. In most cases, the "zero" that results from addition or subtraction will be a negative zero, *e.g.* the sum of $+7305_8$ and -7305_8.

The only difference between one's Complement arithmetic and two's complement arithmetic is the addition of 1 to the low-order bit (bit 1) of the one's complement notation. For example, to the one's complement form of -7305_8, we add 1,

111	*000*	100	111	010
+				1
111	000	100	111	011_2

thus yielding the two's complement form 111 000 100 111 011_2. Positive zero (+0) is the only representation of "zero" in two's complement arithmetic.

For arithmetic purposes, the magnitude value in bits 14–1 is thought of as a fraction, i.e. the binary point is between the sign and bit 14. The magnitude value of bits 14–1 can therefore be thaught of as being in the range $0 - 2^{-14} + 1$. In this "fixed point" arithmetic, the programmer must keep track of the position of the imaginary binary point within a word according to the appropriate scaling, i.*e.* the imaginary binary point is in places to the right of the fixed binary machine point

between bits 15 and 14. For example, if we wish to add the quantities

	bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
		0	.	0	0	0	0	x	0	0	0	0	0	0	0	0
and	+	0	.	0	0	0	0	o	o	o	o	o	y	o	o	o	o	,

where x is scaled 2^6 and y is scaled 2^{10}, we would get no meaningful sum without shifting x to the right 4 bits or y to the left 4 bits.

In multiplication, where there is no need to shift the multiplier or multiplicand right or left, the programmer need only keep track of where the imaginary binary point is in the product, For example,

x . 2^{-4} : bits 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

0 - - - x - -

multiplied by

y . 2^{-2}: bits 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

0 - y - - - - - - - - - - - -

yields z. $2^{-(m+n)}$ or z . 2^{-6}. If we wished the product to be scaled to 2^{-3} or to 2^{-9}, we could either shift the product right or left, or we could have shifted the multiplier or multiplicand before multiplying. In any case, it is important that we be careful not to lose significant bits by shifting a term right or left.

When a word is read out of memory into the 16-bit A Register, or accumulator, the magnitude bits 14 – 1 of the word go into the corresponding bits 14 – 1 of the accumulator. The sign bit goes into bit 15 of the accumulator, from which it is duplicated into bit 16. Carries from bit 14 propagate to bit 15, identified as Sign_1, (S_1), and from S_1 to bit 16, identified as Sign 2 (S_2). The S_2 bit is considered to contain the sign of the word and is the bit sensed to determine the sign of the accumulator quantity.

Under normal conditions, the S_2 and S_1 bits will be equal. In an overflow situation, however, in which bits 14 – 1 are insufficient to define the magnitude of the sum of two terms, S_1 and S_2 will be unequal. For example, the addition of posmax (the maximum positive quantity definable in 14 bits), which is 37777_8, and +1 results in 40000_8, thus causing an overflow into the 15th bit. Since 37777_8 is a positive quantity, bit 16 will be zero, and the configuration of the sum of the two terms will look thus:

bits:

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	1	0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
↑ S_2	↑ S_1														

The S_1 and S_2 bits are unequal, containing the overflow configuration of a 0 in bit 16 and a 1 in bit 15.

Inequality between S_1 and S_2 may also result from a case of negative overflow, For example, let us add negmax (the maximum negative quantity definable within 14 bits), which is -37777_8 or 40000_8 in complemented form, to itself. 40000_8 goes into the accumulator as 140000_8, with a 1 in S_1 and S2 to indicate the negative sign of the quantity. We have

$$\begin{array}{r} 140000_8 \\ + \\ 140000_8 \\ \hline \end{array}$$

which is

$$\begin{array}{r} 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ +\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ +\ \text{carry} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1 \\ \hline +1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1_2 \\ \uparrow S_2\ \uparrow S_1 \end{array}$$

The S_1 and S_2 bits are unequal, containing the negative overflow configuration of a 1 in bit 16 and a 0 in bit 15.

If we now use a TS instruction to store this quantity into memory, the hardware will combine bit 16, the correct sign bit, and the 14 low-order bits (thus bypassing bit 15). The quantity $100\ \ 000\ \ 000\ \ 000\ \ 001_2$ or 40001_8, which is negmax + 1, will be the number stored into memory.

The TS instruction also causes the hardware to look at bit 15, which is still in the accumulator as the uncorrected sign bit, and to set the accumulator cqunl to ± 1 depending upon the configuration of bit 15. Since bit 15 has a zero configuration in this case, the hardware will leave a -1 (77776_8) in the accumulator. In the previous

example of posmax +1, the TS would leave +1 in the accumulator. Thus we may test for overflow, leave the overflow in A, and store the modulus into memory. This procedure is basic to DP operations.

1.4 Instructions

1.4.1 Basic Instructions

A detailed explanation is given below of each of the non-extracode instructions whose code names (with alternate spellings in brackets) are given in the upper half of the chart in Fig. 3. The extracode bit is equal to zero in all of the instructions.

I denotes "at this address." (K) denotes the contents of location K, as distinguished from K, which denotes the address K. $(K)_p$ refers to the previous contents of K, The symbol $\Rightarrow$ denotes "implies." MCT denotes Machine Cycle Time, one MCT being approximately equal to 12 microseconds. The average instruction requires 2 MCT, or 24 microseconds.

The next sequential instruction will always be taken from I + 1 unless specified otherwise. E-memory and F-memory denote, respectively, erasable memory and fixed memory. Since locations $0020-0023_8$ in erasable memory are special registers (see page 5), we edit out any address K where K is 0020—0023 unless otherwise specified, The Assembler gives a diagnostic (also spelled CUSS) as printed output to indicate an assembly error in using the instructions.

TC

Op-Code 00 TC K

K ≠ 3, 4, 6

Transfer Control (to K) 1 MCT

The address K goes into the *Z* Register, and the previous contents of the *Z* Register go into the *Q* Register. Thus the next instruction is taken from K.

Indirect addressing is made possible because the op-code is zero, e. g., the contents of K, which are equal to 0/XXXX where XXXX is some address, become TC XXXX.

Since TC K may be used as a subroutine call, it is obviously necessary to preserve the contents of the *Z* Register in the Q Register *so* that we may return to the main program upon completion of the subroutine. Since a TC causes the previous contents of the *Z* Register to go into Q, Q is now pointing at "TC + 1" and a TC *Q* at the end of a subroutine will return control indirectly to the place at which we had originally quit the main program. For this type of operation, TC may be spelled TCR, for Transfer Control Setting up Return.

TC A will obviously transfer control to the contents of the accumulator, where the op-code configuration of the accumulator bits will determine the next operation to be performed. If the programmer has made sure that bits 15—13 of the contents of A are equal to zero, control will be indirectly transferred to whatever address bits 12—1 of the accumulator specify.

Special cases of TC K occur when the address K is equal to 3, 4, or 6. In these three cases, the indicator specified by K is set, and the next instruction is taken from I + 1.

TC 3 = RELINT (Allow Interrupt)

TC 4 = INHINT (Inhibit Interrupt)

TC 6 = EXTEND (Set Extracode Switch)

The extracode switch causes the next instruction to be an extracode. As we have said above, any instruction except "INDEX" resets the switch. Interrupt is inhibited while the switch is on. Other uses of "TC" will be discussed under Special Codes.

CCS

Op-Code 01 **CCS** K (K must be in erasable)

QC 0 Count, Compare, and Skip 2 MCT

(Quarter Code)

I

1 + 1

I + 2

I + 3

I + 4

(K) go into the accumulator.

If (K) > 0, then we take the instruction at I + 1, and (A) will be reduced by 1, i. e. (K) - 1. If (K) = + 0, we take the instruction at I + 2, and (A) will be set to +0. If (K) < -0, we take the instruction at I + 3, and (A) will be set to its absolute value less 1. If (K) = -0, we take the instruction at I + 4, and (A) will be set to + 0. CCS always leaves a positive quantity in A.

This is the only compare instruction. It is also used for loop control and indicator testing. For example, if we wish to transfer out af a subroutine

when the positive contents of the accumulator become zero, a "CCS" will cause the contents of A to be reduced by 1. When (A) reaches + 0, a TC K placed two instructions after the CCS will cause the desired transfer out of the subroutine.

Also, we might wish to use a 1 configuration in bit 9 of some word which is ordinarily set to 0 to indicate that jets should be turned on to propel the spacecraft in some direction. We would use a CCS A after isolating bit 9 in A to test (A) for a quantity greater than + 0. If we found a quantity > + 0, we would branch to a sequence of instructions controlling the operation of the jets.

TCF

Op-Code 1 TCF K (K must be in F-memory)
QC ≠ 0

Transfer Control to Fixed Memory 1 MCT

Take the next instruction from K and proceed from there. Using TCF rather than TC is a convenient way of having the assembler do address checking for you. If the operand, K, is not in fixed memory, a diagnostic will come out of the assembler. Moreover, TCF does not change the Q Register.

DAS

Op-Code 2 DAS K (K must be in E-memory)
QC = 0 Note: this assembles as DAS K + 1

Double Add to Storage (to and from K) 3 MCT

The contents of the accumulator and its L Register are added to the contents of K and K + 1. The DP sum is stored back into K and K + 1. If positive (negative) overflow results from the DP addition, the sum is stored into K and K + 1, and the net overflow (+1 if positive; -1 if negative) is left is left in the A Register. If no overflow resulted, +0 would be left in the A Register. +0 is left in the L Register.

DAS A doubles the contents of the DP accumulator. (The assembly mnemonic DDOURL assembles as DAS A).

LXCH

Op-Code 2 LXCH K (K must be in E-memory)
QC = 1

Exchange L and K 2 MCT

The contents of the L Register are exchanged with the contents of K. We could LXCH A, in which case A would be overflow-corrected before the swap.

For example, the instruction LXCH 1037_8 would cause the contents of location 1037_8 to go into the L Register and the contents of location 1037_8 to be replaced by the previous contents of the L Register.

INCR

Qp-Code 2 INCR K (K must be in E-memory)
QC = 2

Increment (K) 2 MCT

The contents of K are replaced by the contents of K incremented by 1. A is not affected.

NOTE: INCR and two other codes AUG and DIM are modified counter-increment sequences. Thus, if one of these three overflows when addressing a counter for which overflow during involuntary incrementing is supposed to cause an interrupt, the interrupt will occur. This is also true for chain-reaction increments like T2, which is incremented after an overflow of T1. These three instructions INCR, AUG, and DIM always operate in one's complement arithmetic, even when addressing CDU counters, which normally use two's complement arithmetic,

ADS

Op-Code 2 ADS K (K must be in E-memory)
QC a 3

Add to Storage 2 MCT

The contents of the accumulator and the contents of K are replaced by the sum of the contents of the accumulator and of K.

Let us consider the instruction ADS 2074_8 where location 2074_8 contains the quantity 7. The instruction causes 7 to be added to the contents of the accumulator. The sum will replace both 7 and the previous contents of the accumulator. Location 2074_8 now contains (7 t A p). The overflow-corrected result is always stored, but overflow, if it occurred, would remain in A.

CA

Op-Code 3 CA K

Clear and Add (K) 2 MCT

The contents of K come into the accumulator, leaving the contents of K unchanged. Alternate spelling CAF (Clear and Add Fixed) or CAE (Clear and Add Erasable) may be used when referencing fixed or erasable memory, if assembler-checking of the addresses is desired.

For example, CAF 4671_8 would clear the accumulator, and the contents of location 4761_8 would be duplicated into the accumulator. If we had said CAE 4761_8, we would have received an assembly diagnostic.

CS

Op-Code 4 CS K

Clear and Subtract (K) 2 MCT

The one's complement of K comes into the accumulator, leaving the previous contents of K unchanged.

For example, if location 34_8 contained the quantity 22, the instruction CS 34_8 would clear the accumulator and 77755_8 would come into A. The contents of location 34_8 would still be 22.

TS

Op-Code 5 TS K (K must be in E-memory)

QC 2

Transfer to Storage

The contents of the accumulator, bits 16, 14—1 come into K. If there is positive or negative overflow in the previous contents of the accumulator, we set the contents of the accumulator equal to plus or minus one,

respectively, and take the next instruction from I + 2. If no overflow exists, take the next instruction from I + 1.

TS A guarantees that the contents of the accumulator will be equal to the previous contents of the accumulator, but if overflow existed in A it causes us to skip to I + 2 for the next instruction. OVSK (Overflow Skip) is the implied-address code for this use of TS.

Consider the instruction TS 0043_8. The previous contents of the accumulator are stored into location 43_8, and the contents of A are replaced by a ± 1 if there was positive or negative overflow in the previous contents of A,

INDEX

Op-Code 5 INDEX K (K must be in E-memory)
QC = 0

Index Next Instruction 2 MCT

See also Op-Code 15, page 42

One basic idea of indexing is to provide a method which will enable us to access some element within a list which has starts at a "base" address. For example, assume that we have a list of 100 elements in memory with a base address called TABL1, and assume that we wish to bring random elements into the accumulation for various operations. Since TABLl is the only address we have to reference the 100 elements, indexing allows us to address $TABL1_i$, where i is the address of the element relative to the base address. The instruction CAE $TABL1_7$ would clear the accumulator and bring into it the contents of the 7th element in TABL1.

This method of modifying the base address is usually accomplished through the use of index registers, which contain the quantity to be added to or subtracted from the base address to access some element within the list. With the AGC, however, we use an INDEX instruction to perform the function of an index register.

INDEX K: The contents of K are added, bit by bit, to the next sequential instruction. (We shall take this sum and execute it as the next instruction. The contents of K remain unchanged as do the contents of the next instruction. The Z Register is set to access the second instruction beyond the INDEX instruction).

If the contents of K are equal to or less than 1777_8, we have just augmented the address portion of the next instruction, For example, if we say

INDEX	K [(K) = 55_8]
CAE	TABL1,

we have modified the second instruction to read

(CAE TABL1 + 55),

which means that we shall replace the previous contents of the accumulator with the contents of the 55_8 th element in TABL1.

If the contents of the address K are equal to or less than 7777_8 but greater than 1777_8, we may change the quarter-code or the Effective Operand Address to indicate the need for a bank (when none was called for by the original sequential instruction). The reason for this is that, unlike the index register, which modifies only the address portion of an instruction, INDEX K causes the contents of the address K be added to the entire following instruction. Thus we may change the quarter code bits which serve as indicators for memory banks, and we may change bits 15—13to indicate an altogether different operation. For example, INDEX A causes the contents of the accumulator to be added to the next word in memory. If the next instruction was 000 000 000 000 000, we would then be executing the contents of A (which is the equivalent of the instruction XXALQ or TC A). If the next instruction was not 000 000 000 000 000, we have, in effect, modified the contents of its address field or its op-code, or both (because of overflow into the op-code field).

Another common use of indexing is to control a loop through an area of memory. Assume we have some positive count in a counter. As long as this count remains positive, we wish to continue some operation, which we shall terminate when the count reaches + 0. Specifically, let us program the problems as seen in Fig. 4 and Fig. 5

1.4.2 Special Codes

INDEX 17_8 is a special use of INDEX, and means Resume Interrupted Program (which like INDEX K, requires 2 MCT). The contents of the Z Register are set to the contents of location 15, and the next instruction is taken from the contents of location 17. The implied-address code RESUME assembles as INDEX 17. INDEX 17_8 does not mean INDEX location 17_8. The interrupt processing, in which this instruction is used, will be discussed later.

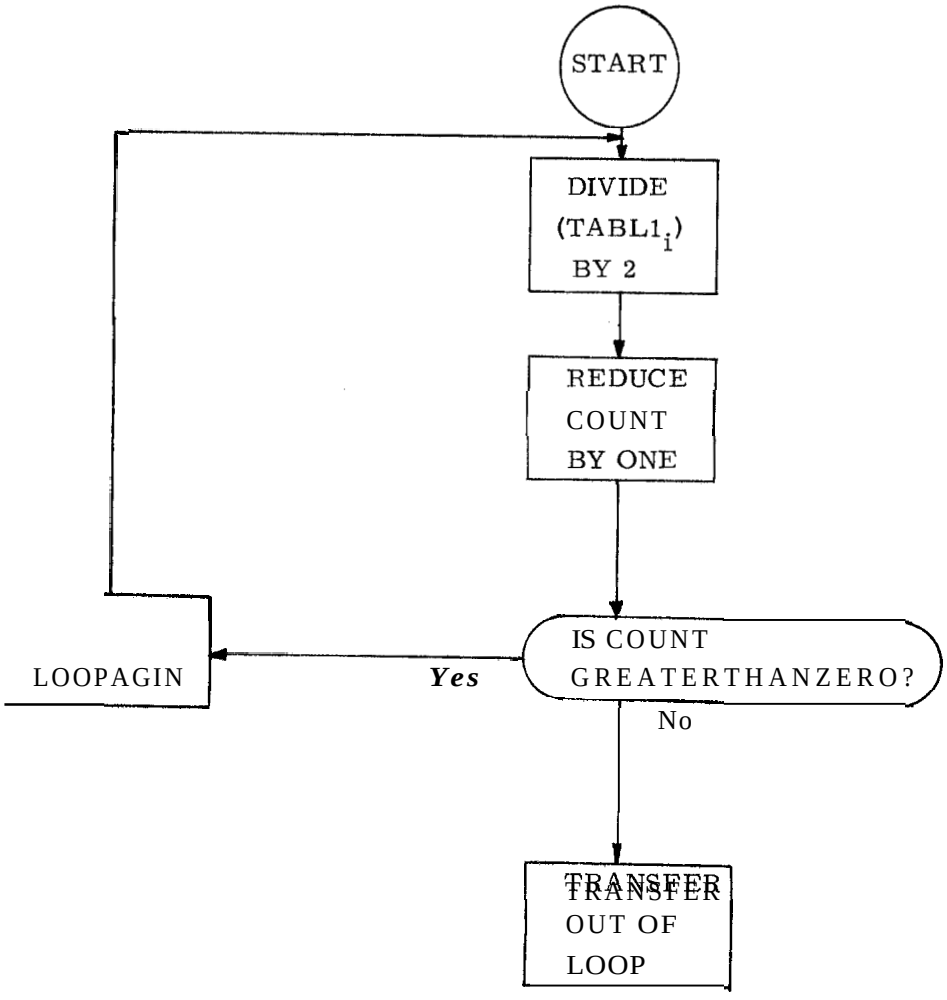

START
DIVIDE
(TABL1$_i$)
BY 2
REDUCE
COUNT
BY ONE
IS COUNT
GREATERTHANZERO?
Yes
LOOPAGIN
No
TRANSFER
OUT OF
LOOP

Figure 4

ROUTINE TO HALVE 10 ELEMENTS OF TABL 1

(TABL 1_0 - TABL 1_9)

LOC	OP-CODE	OPERAND	COMMENT
START	CA	NINE	ENTER HERE
LOOPAGIN	TS	COUNT	INITIALIZE COUNTER TO t 9*
	INDEX	COUNT	COUNT GOES FROM 9 to *0*
	CA	TABL 1	GET TABL 1 "(COUNT)
	TS	SR	DIVIDE (TABL 1_i) by 2
	CA	SR	$\frac{(\text{TABL } 2_i)}{2}$ A
	INDEX	COUNT	
	TS	TABL 1	(A) TABL 2_i +(COUNT)
LOOPTEST	CCS	COUNT	NOW TEST IF COUNT = 0
	TCF	LOOPAGIN	NO: A NOW HAS (COUNT) p - 1
	TCF	EXIT	LEAVE LOOP
NINE	DEC	9	CCS will never come here in this case,
COUNT	DEC	*0*	TEMPORARY so these words may be used in any way.

* Subsequently, set (COUNT) = (COUNT) p - 1

Figure 5

INDEX 17 and three other instructions: TC **3,** TC 4, and TC 6, introduced on page 1-22 under TC are special codes:

TC **3:**	INHINT
TC **4:**	RELINT
TC 6:	EXTEND
INDEX 17:	RESUME

The combination of the contents of the address of an INDEX instruction and the instruction following the INDEX will never result in any one of the special codes. The reason for this lies in the following hardware scheme:

1.4.3 Op-Code Selection Logic

We may get the instructions

```
TC 3
TC 4
TC 6
INDEX 17
```

by preceding some instruction with an INDEX K, but the instruction will be interpreted literally as

```
TC location 3
TC location 4
TC location 6
INDEX location 17.
```

For example, if we say

```
INDEX K
TC 0
```

where K contained 4, we would execute the instruction TC **4** literally.

Neither could we get the special codes INHINT, RELINT, EXTEND, or RESUME if we precede any instruction with an EXTEND instruction. EXTEND sets the extracode bit to 1, which immediately prevents us from getting any instruction other than an extracode. The special codes are all non-extracodes. For example, if we say

```
EXTEND
RELINT
```

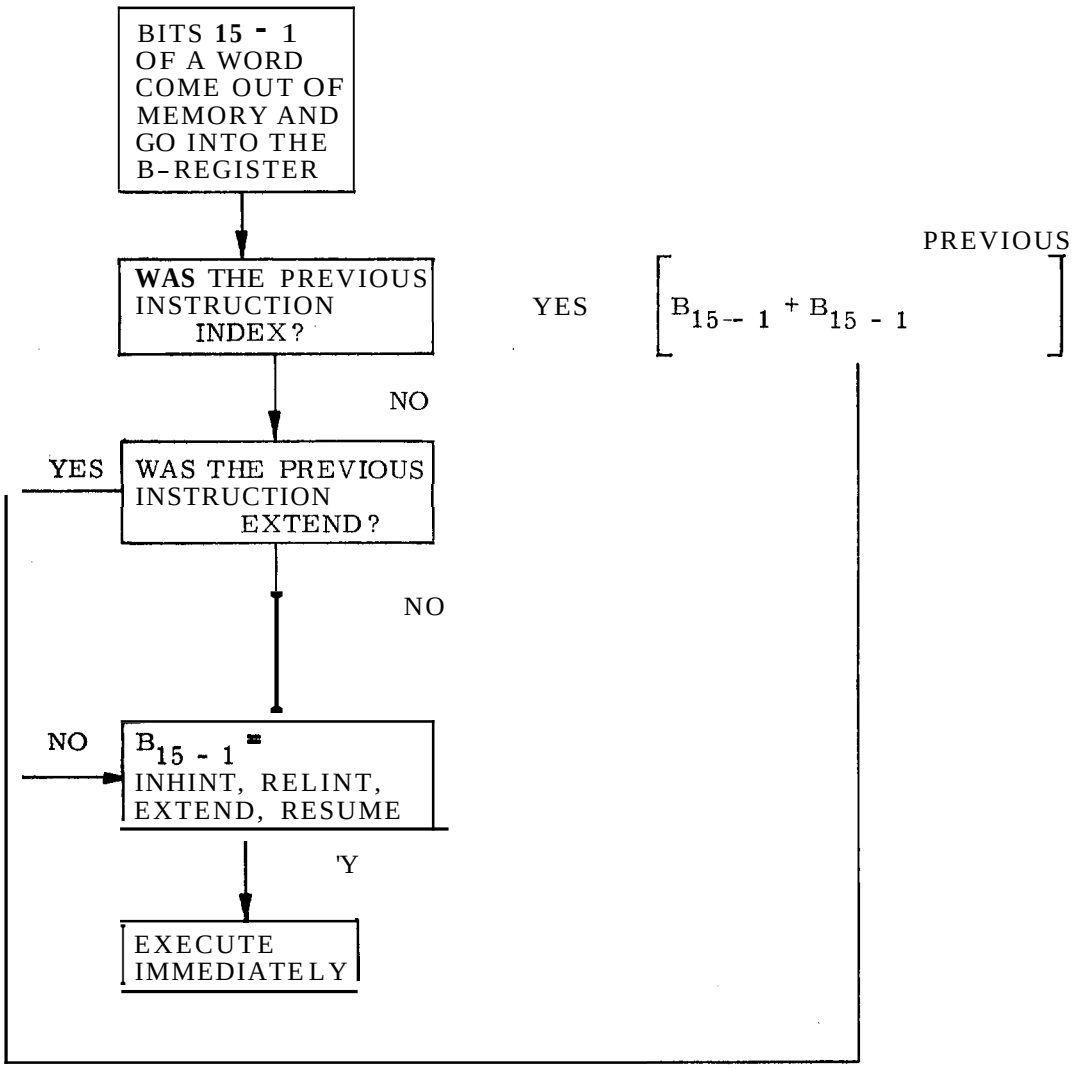

BITS 15 - 1 OF A WORD COME OUT OF MEMORY AND GO INTO THE B-REGISTER
WAS THE PREVIOUS INSTRUCTION INDEX?
YES
PREVIOUS
B15-- 1 + B15 - 1
NO
YES
WAS THE PREVIOUS INSTRUCTION EXTEND?
NO
NO
B15 - 1 = INHINT, RELINT, EXTEND, RESUME
'Y
EXECUTE IMMEDIATELY

Figure 6

we would get a channel instruction (look at the instruction diagram on page 1-31). Likewise, if we say

EXTEND
RESUME

we would execute the instruction INDEX 17 (not RESUME).

RELINT

Op-Code 00 RELINT
K = *0003*

Release (allow) Interrupt 1 MCT

Allow interrupt after this instruction (subject to the restriction that interrupt cannot occur while there is positive or negative overflow in the accumulator, nor between an interrupt and a subsequent RESUME.

INHINT

Op-Code 00 INHINT
K = 0004

Inhibit Interrupt 1 MCT

Inhibit interrupt until a subsequent RELINT. The inhibition set by INHINT and removed by RELINT is entirely independent of the one set by interrupt and removed by RESUME. Either one alone is sufficient to prevent interrupt,

EXTEND

Op-Code 00 EXTEND
K = 0006

Extend Next Instruction 1 MCT

Take the next instruction from I + 1 and execute it as an extracode (i.e. set the extracode bit of the next instruction to 1). If the next instruction is INDEX, the instruction following the INDEX will be executed as an extracode instruction too.

RESUME

Op-Code 5 RESUME
QC 0
K = 17

Resume Interrupted Program 2 MCT

The contents of location 15 come into the Z Register, replacing the previous contents of Z. Use the contents of location 17 as the next instruction. Allow interrupt after this instruction (unless there has been an INHINT with no following RELINT).

1.4.4 Remaining Basic Instructions

DXCH

Op-Code 5	DXCH	K (K must be in E-memory)
QC 1		
	Double Exchange	3 MCT

The contents of K are exchanged with the contents of the accumulator, and the contents of K + 1 are exchanged with the contents of the L Register. The final contents of the L Register will be overflow-corrected. The operation code should be treated as 52001_8 (See Note, Page 1-23).

XCH

Op-Code 5	XCH	K (K must be in E-memory)
QC 3		
	Exchange A and K	2 MCT

The contents of the accumulator are exchanged with the contents of K.

If location 1730_8 contained the quantity 7, for instance, the instruction XCH 1730_8 would cause 7 to come into the accumulator and the previous contents of the accumulator to go into location 1730_8.

AD

Op-Code 6	AD	K
	Add (K)	2 MCT

The contents of K are added to the contents of the accumulator, and the sum is stored back into the accumulator, The contents of K remain unchanged.

If the accumulator contained the quantity 30_8, and the quantity 10_8 was in location 4441_8, the instruction AD 4441_8 would cause 40_8 to replace the previous contents of the accumulator. Location 4441_8 would still contain the quantity 10_8.

MASK K

Op-Code 7 MASK K

Mask A by K 2 MCT

This instruction causes a logical AND operation. The contents of K are "ANDed" with the contents of the accumulator, and the result replaces the previous contents of the accumulator. The symbol $\wedge$ denotes the logical AND function. The truth table for each bit position of the contents of A and K is as follows:

A	K	A ∧ K
0	0	0
0	1	0
1	0	0
1	1	1

Mask 0020 — 0023 does not result in re-editing; i. e. (K) = (K)p.

1.4.5 Extracode Instructions

These instructions require the use of a 16th bit, called the extracode bit, for op-code definition. Since the extracode bit is set by an EXTEND instruction, all extracode instructions must be preceded by an EXTEND instruction. Any instruction other than an INDEX instruction will reset the extracode bit to 0.

EXTEND plus a zero op-code (i. e. 0 — 10) plus bits 12, 11, and 10 is broken down into seven peripheral codes (PC 0 — PC 6). Each uses a 9-bit address to reference an input-output channel (KC). The A and L Registers are channels 0 and 1, respectively, to facilitate complicated logic in an arithmetic register.

READ

Op-Code 10 READ KC

PC 0

Read Channel KC 2 MCT

The contents of channel KC, where KC is an in/out channel, come into the accumulator, replacing the previous contents of the accumulator.

For example, READ 1 would cause the contents of channel 1, which is the L Register, to come into the accumulator,

WRITE

Op-Code 10 WRITE KC
PC 1

Write Channel KC 2 MCT

The contents of the accumulator come into KC, replacing the previous contents of channel KC.

RAND

Op-Code 10 RAND KC
PC **2**

Read and Mask **2** MCT

The contents of the accumulator are ANDed with the contents of channel KC, and the sum is stored back into the accumulator, replacing the previous contents of A. The symbol Λ denotes the logical AND function *(see* MASK, page **1-34).**

WAND

Op-Code 10 WAND KC
PC **3**

Write and Mask 2 MCT

The contents of the accumulator are ANDed with the contents of channel KC, and the sum is stored back into channel KC and is duplicated in- ' to the accumulator.

ROR

Op-Code 10 ROR KC
PC **4**

Read and Superimpose 2 MCT

The contents of the accumulator are ORed with the contents of channel KC. The symbol v denotes the logical OR function. The truth table for each bit position of the contents of the accumulator and of the channel KC is as follows:

A	KC	A v KC in A
0	0	0
0	1	1
1	0	1
1	1	0

WOR

Op-Code 10 WOR KC
PC 5

Write and Superimpose 2 MCT

The contents of the accumulator are ORed with the contents of channel KC. The sum is stored back into channel KC and is duplicated into the accumulator, replacing the previous contents of both A and channel KC.

RXOR

Op-Code 10 RXOR KC
PC 6

Read and Invert 2 MCT

The contents of the accumulator are Exclusive ORed with the contents of channel KC, The symbol ⊻ denotes the logical Exclusive OR function. The truth table for each bit position of the contents of the accumulator and of the channel KC is as follows:

A	KC	A ⊻ KC
0	0	0
0	1	1
1	0	1
1	1	0

This logical function is used to invert bit settings in channels for input-output. For example, bit 5 in channel 7 controls the KEY RELEASE light on the Display-Keyboard (DSKY). When bit 5 contains a zero, the KEY-RELEASE light is off, and remains off if we pulse another zero into bit 6. When bit 5 is set to 1, the KEY RELEASE light blinks on and off, and it continues to blink if we pulse another 1 into bit 5. The behavior of the KEY-

RELEASE light will change only when bit 5 contains a zero and we pulse a one into it, causing the light to start blinking, or when bit 5 contains a one and we pulse a zero into it, causing KEY–RELEASE to stop blinking.

DV

Op-Code 11 DV K
QC = *0*

Divide (by K) 6 MCT

The contents of the accumulator and of the L Register, the dividend, are divided by the contents of K, the divisor, leaving the quotient in A and the remainder in L.

We determine the sign of the quotient by the usual arithmetic law of combining the signs of the dividend and divisor. Since the signs of the double-length dividend in A and L need not agree, we understand the final sign of the dividend to be the sign of the accumulator, unless the contents of the accumulator are plus or minus zero. In this latter case, the sign of the dividend will be the sign of the L Register. The remainder bears the sign of the dividend, determined as discussed above.

The instruction DV does not disturb the contents of the Q Register and does not re-edit an argument between 0020 – 0023.

We may divide only a larger number into a smaller number, i. e. the divisor must be larger than the dividend.

If a quantity is divided into a quantity of equal magnitude, we get a quotient of either posmax or negmax and a remainder equal to the dividend. If a smaller is divided into a larger quantity, however, we get total nonsense which cannot be distinguished from significant data. No alarm light flashes, and the machine sends forth no diagnostic. Scaling may therefore be necessary to assure a legal divide and to properly position the scale factor of quotient. Scaling may also be necessary to guarantee maximum precision to your answer.

Consider the instruction DV 00015_8 preceded by an EXTEND instruction. Assume location 00015_8 contains the quantity **4** and the accumulator

contains the quantity 10_8. Since the divisor, 4, scaled 2^{-14}, is smaller than the dividend (the accumulator quantity 10_8, also scaled 2^{-14}), we must scale the contents of 00015_8 to 2^{-12} by shifting it left two places. We may now legally divide the contents of the accumulator and the L Register by the contents of 15, giving us

$$2^{-14} \times 10_8 = 2 \times 2^{-2}$$

$$2^{-12} \times 20_8$$

which looks like

	A						L				
	0	001	*000*	*000*	*000*	000	*000*	*000*	*000*	*000*	*000*
(bits)	16	15 . , ...				1	15	14 . . . , ...			1

BZF

Op-Code 11 BZF K (K must be in F-memory)

QC ≠ 0

Branch Zero to Fixed 1 or 2 MCT

If the contents of the accumulator are equal to positive or negative zero, take the next instruction from K, and proceed from there (1 MCT). Otherwise, take the next instruction from I +1 (2 MCT).

For example, assuming the quantity -777 was in the accumulator, we take the next instruction from I + 1 on the instruction BZF 4305_8. If positive zero or negative zero was in the accumulator, the instruction BZF 4305 would cause us to take the next instruction from location 4305, and proceed from there.

MSU

Op-Code 12 MSU K

QC 0

Modular Subtract 2 MCT

The contents of K are modular subtracted from the contents of the accumulator, and the difference is stored back into the accumulator. The contents of K remain unchanged.

The symbol θ denotes modular subtraction, which forms a signed one's complement difference of two unsigned (modular or periodic) two's complement inputs. The method is to form the two's complement difference, to decrement it if it is negative, and to take the overflow-uncorrected sum as the result.

For example, consider the modular subtraction of 30000_8 (135^O) from 20000_8 (90^O), the quantity in the accumulator. We take the two's complement form of 30000_8, which is 147777 (duplicated sign)

$$\begin{array}{r} + \quad 1 \\ \hline 150000_8 \end{array}$$

and add to it the quantity in the accumulator.

$$\begin{array}{r} 1.\,50000 \\ \textit{to.}\ 20000 \\ \hline 1.70000_8 \end{array}$$ whose binary configuration in A is

	1	111	000	*000*	*000*	*000*
bits	16	15.				1

Since this is in two's complement notation, we convert it to one's complement by subtracting one from it (adding the one's complement of one).

$$\begin{array}{r} 170000 \\ +177776 \\ \hline 167776 \\ + \quad 1 \\ \hline 1.67777_8 \end{array}$$

which represents the one's complement of -10000 (or -45^O).

We may also do this problem by adding the one's complement form of the contents of K to the contents of the accumulator,

$$\begin{array}{r} 20000 \\ +47777 \\ \hline 67777 \end{array}$$

and test the sign bit, If bit 15 is a one, as in this case, we add a one to the sum, thus giving us

$$\begin{array}{r} 67777 \\ +\quad 1 \\ \hline 70000_8, \end{array}$$

which, when converted to one's complement form, gives us -10000, as before. If bit 15 contained a zero, we would not add a one to the sum.

CDU counters keep track of the gimbal angles of the inertial measurement unit and optics unit in two's complement notation, We take the difference between what the gimbal angles are and what we wish them to be, and use this difference to drive the CDU's. Since the AGC uses only one's complement arithmetic, we use the modular subtraction instruction to resolve the problem of having a one's complement computer and two's complement counters.

QXCH

Op-Code 12 QXCH K (K must be in E-memory)
QC = 1

Exchange Q and K 2 MCT

The contents of K are exchanged with the contents of Q.

Q may contain a return address after TC. For example, when we leave a main program to execute a subroutine, the Q Register will contain the instruction in the main program to be executed immediately after completing the subroutine. To transfer out of the subroutine, then, we just say TC K, and we shall resume the main program. A prior QXCH K saved (Q) in K and freed Q for use.

AUG

Op-Code 12 AUG K (K must be in E-memory)
QC = 2

Augment 2 MCT

If the contents of K are equal to or greater than + 0, we increment the contents of K by 1 and store it back into K. If the contents of K are equal to or less than -0, the contents of K are decremented by 1, and the result is stored back into K.

For example, if K contained 7, we would increment 7 by 1, leaving 10_8 in K. If K contained -7, we would decrement -7 by 1, leaving -10_8 in K.

See NOTE, page 1-24.

DIM

Op-Code 12 DIM K (K must be in E-memory)
QC = 3

Diminish

If the contents of K are greater than + 0, we decrement the contents of K by 1 and store the result back into K. If the contents of K are less than -0, we increment the contents of K by 1 and store the result back into K.

If the contents of K are equal to plus or minus zero, we leave the contents of K unaltered. A is not changed.

See NOTE, page 1-24.

DCA

Op-Code 13 DCA K
Double Clear and Add 3 MCT

The contents of K come into the accumulator, and the contents of K + 1 come into the L Register. The contents of K and of K + 1 remain unchanged. The final contents of the L Register will be overflow-corrected. This instruction assembles as DCA K + 1.

For example, the instruction DCA 7001_8 would first clear the accumulator and the L Register of their previous contents. The contents of location 7000_8 would then go into the accumulator, and the contents of the next location 7001_8 would go into the L Register. The contents of K and of K +1 would remain unchanged.

DCS

Op-Code 14 DCS K
Double Clear and Subtiract (K) 3 MCT

The one's complement of the contents of K come into the accumulator, and the one's complement of the contents of K +1 come into the L Register. The contents of K and K + 1 remain unchanged. The instruction DCS K assembles as DCS K + 1.

DCS A complements the double precision accumulator; the implied-address code is DCOM. The final contents of the L Register will be overflow-corrected,

Consider the instruction DCS 22223, which would first clear the accumulator and L Register. The one's complement of the contents of location 22222 would go into the accumulator, and the one's complement of the contents of the next location 22223_8 would go into the L Register. The contents of locations 22223_8 and 22222_8 would be left unaltered.

INDEX

Op-Code 15	INDEX	K (anywhere in memory)
	Index Extracode Instruction	2 MCT
	(see INDEX, page 1-26)	

This is the only extracode instruction that does not reset the extracode switch. The way to index an extracode (e.g. MP) is

```
EXTEND      ADDRWORD
INDEX       0
MP
```

The extracode switch will be maintained through any n-level nesting of extracode INDEX's. This is logical, since INDEX does not reset the extracode bit to zero. We can, therefore, precede an instruction with any number of INDEX's without losing our extracode bit setting.

This INDEX will never form a special op-code instruction (see INDEX, page 1-27).

SU

Op-Code 16	SU	K (K must be in E-memory)
QC = 0		(K)
	Subtract	2 MCT

The contents of K are subtracted from the contents of the accumulator, and the difference is stored back into the accumulator. The contents of K remain unchanged. Overflow may result,

BZMF

Op-Code 16 BZMF K (K must be in F-memory)

QC ≠ 0

Branch Zero or Minus to Fixed 1 or 2 MCT

If the contents of the accumulator are equal to or less than positive zero, take the next instruction from K and proceed from there (1 MCT). Otherwise, take the next instruction from I +1 (2 MCT).

MP

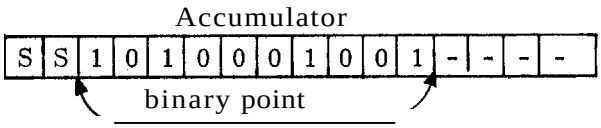
Op-Code 17 MP K

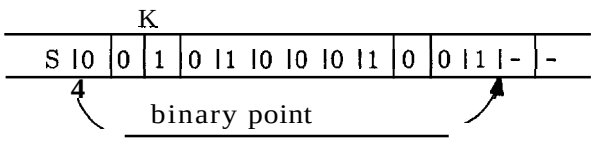
Multiply 3 MCT

The contents of the accumulator are multiplied by the contents of K. The product is stored back into the accumulator and the L Register, and the sign of the product is formed by the rules of algebra.

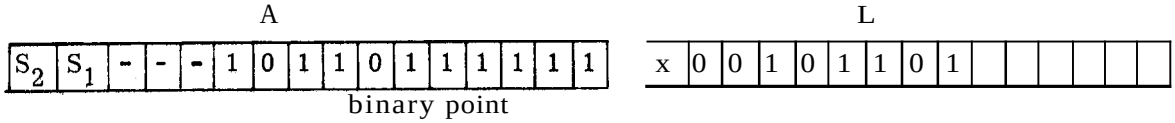
The two words *of* the product agree in sign. A zero result is positive (unless the contents of the accumulator were equal to positive or negative zero, and the contents of K are non-zero with the opposite sign). MP does not re-edit an argument from 0020—0023.

Scaling may be necessary to assure sufficient and/or maximum precision in the product, For example, if we multiply 1211_8, scaled 2^{-10}, by 1211_8, scaled 2^{-12},

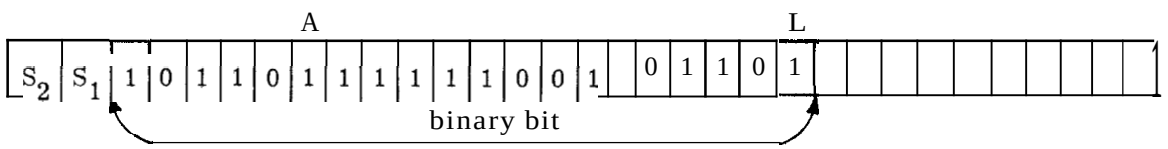




we get 376621_{10}, or 1337455_8, scaled 2^{-22}



Since the product 1337455_8 actually requires only 19 bits for definition and since we have positioned it in A and L according to its scale factor 2^{-22}, $\left[(1211_8 \times 2^{-10}) \cdot (1211_8 \times 2^{-12}) = 1337455_8 \times 2^{-22}\right]$, we know that the three high-order bits in A are leading zeroes, Thus we can shift the product to the left three places in order to have as many meaningful bits as possible in A without losing any significant high-order bits.



We now have as many meaningful bits as possible in the more significant of the two product registers, as seen above.

1.4. 6 Implied-Address Codes

Certain instructions, like RESUME, are defined for only one address value, and others have unusual results when used to address special registers. For convenience in using these instructions, the YUL System assembler recognizes implied-address codes written without an address, and it fills in the address. These codes are shown alphabetically on Page 1-45. Some of these codes arise from the fact that certain special registers are adjoining locations in erasable memory:

Location	Register
0	A (accumulator)
1	L (low register)
2	Q
3	EB (erasable bank register)
4	FB (fixed bank register)
5	Z (program counter)
6	BB (both banks register)

Below is an explanation of each implied-address code except INHINT, RELINT, EXTEND, and RESUME, discussed above under Special Codes on Page 1-27.

XXALQ

Op-Code 0 XXALQ

K = 0 (TC A)

Execute Extracode Using A, L, and Q — 2 MCT

Implied Address Codes

Implied Address Code	Actual Operation Code	Register (If applicable)	Word as Assembled	NOTE
COM	CS	A	40000	
DCOM	DCS	A	40001	X
DDOUBL	DAS	A	20001	
DOUBLE	AD	A	60000	
DTCB	DXCH	Z, & BB	52006	
DTCF	DXCH	FB, & Z	52005	
EXTEND	TC		00006	S
INHINT	TC		00004	S
NOOP	TCF		+1 (I + 1)	F
NOOP	CA	A	30000	E
OVSK	TS	A	54000	
RELINT	TC		00003	S
RESUME	INDEX	BRUPT	50017	R
RETURN	TC	Q	00002	
SQUARE	MP	A	70000	X
TCAA	TS	Z	54005	
XLQ	TC	L	00001	
XXALQ	TC	A	*00000*	
ZL	LXCH		22007	
ZQ	QXCH		22007	X

NOTE EXPLANATION:

E Applies when I (location of instruction) is in erasable memory.

F Applies when I is in fixed memory.

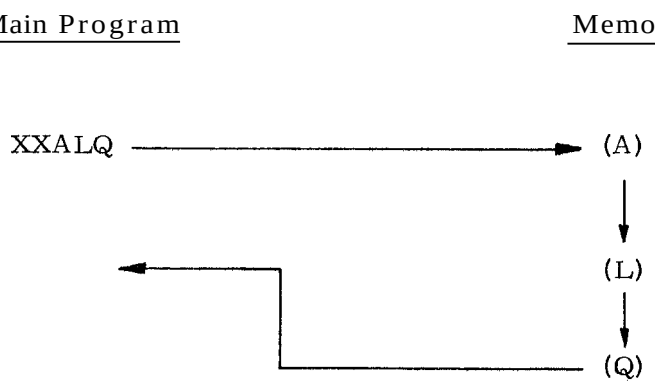
R Special RESUME hardware responds *to* address 0017.

S Special Indicator-setting hardware responds to addresses *0003*, 0004, and 0006.

X Extracode instruction.

Figure 7

Assume that the accumulator contains 00006 (EXTEND) and that L contains an extracode instruction. TC will set Q to contain the 12-bit address of the next instruction to be executed in the main program, XXALQ causes the machine to transfer control (TC) to A, location 0. The EXTEND instruction in the accumulator is executed; then the extracode instruction in L, the next location; and finally the (TC) contents of Q, the next location, which returns control to the main program.



XLQ

Op-Code *0* XLQ

K = 1 (TC L)

Execute Using L and Q 2 MCT

Assume that L contains a basic instruction. Execute the instruction in L, and if it is not a successful branch, return to I + 1.

The time (2 MCT) for XXALQ and XLQ includes the TC A or L and the return TC from Q, but it does not include the time spent in executing the contents of A or L.

RETURN

Op-Code 0 RETURN

K – 2 (TC Q)

Return from Subroutine 2 MCT

Assume that the contents of Q contain the instruction TC K. Take the next instruction from K and proceed from there.

NOOP

Op-Code 1 NOOP

QC ≠ 0 (TCF + 1)

K = I + 1

No Operation (in Fixed Memory) 1 MCT

Take the next instruction from I +1. NOOP is assembled TCF + 1 in fixed memory,

DDOUBL

Op-Code 2 DDOUBL

QC 0

K = 0 (DAS A)

Double Precision Double 3 MCT

The contents of the accumulator and L Register are added to itself, and the sum is stored back into A and L, replacing their previous contents. If the previous contents of the accumulator contained positive or negative overflow, the results are messy, e. g. when the sign of the sum of the DP addition stored in A is unequal to the sign of the previous contents of A. If the previous contents of A were equal to or greater than 1/2, overflow will be retained in the contents of A.

ZL

Op-Code 2 ZL

QC 1

K = 7 (LXCH 7)

Zero the L Register 2 MCT

Zeroes come into the L Register, replacing the previous contents of L.

This code and its companion ZQ depend on two properties of address 0007: no storage is associated with it, and references to it (in fact, to any of 0000—0007) are not checked for good parity. Address 0007 is therefore a generally good source of zeroes.

NOOP

Op-Code 3 NOOP

K = O

No Operation (in Erasable Memory) 2 MCT

NOOP is assembled as CA A in erasable memory.

COM

Op-Code 4 COM
K = 0 (CS A)
Complement (the contents of A) 2 MCT

The one's complement of the contents of the accumulator replaces the previous contents of A, All 16 bits of A are complemented.

QVSK

Op-Code 5 QVSK
QC 2
K = O (TS A)
Overflow Skip 2 MCT

Do not change the contents of the accumulator. If the contents of A contain positive or negative overflow, take the next instruction from I +2. If no overflow exists in the contents of A, take the next instruction from I + 1.

For example, let us clear and add the contents of COUNTER into the accumulator. Suppose we now add the contents of CUM2 to the contents of A. If this addition operation caused either positive or negative overflow in A, we would leave the contents of A unchanged by an OVSK and skip the next sequential instruction, thus taking the next instruction from I + 2. If the addition caused no overflow in A, we would leave the contents of A unaltered and merely take the next sequential instruction.

TCAA

Op-Code 5 TCAA
QC 2
K = 5 (TS Z)
Transfer Control to the Address in A 2 MCT

Bits 12—1 of the contents of the accumulator come into the Z Register. If there is positive or negative overflow in the contents of the accumulator, the contents of A are set to +1 if the overflow is positive and to -1 if the overflow is negative. We take the next instruction from the address specified in Z, as usual.

For example, suppose the contents of the accumulator are

A

0	1	0	0	1	1	0	1	0	0	1	1	0	1	1	0

bits 16 15 12 1

The contents of bits 12—1 come into Z,

Z Register

1	1	0	1	0	0	1	1	0	1	1	0

bits 12 1

and since the accumulator contains positive overflow (overflow because bits 16 and 15 differ, and positive because we take the sign of the word from bit 16, [here containing 0]) we clear A and set its contents equal to +1. We then take the next instruction from the location specified in the Z Register. In this case, we would take the next instruction from location 6466_{8}.

DOUBLE

Op-Code 6 DOUBLE
K = 0 (AD A)

Double (the contents of A) 2 MCT

The contents of the accumulator are added to itself, and the sum is stored back into A.

See remarks on overflow under DDOUBL.

ZQ

Op-Code 12 ZQ
QC 1
K = 7 (QXCH)

Zero Q 2 MCT

Positive zeroes replace the contents of the Q Register. (See the discussion under the instruction ZL).

DCOM

Op-Code 14 DCOM
K = 0 (DCS A)

Double Complement 3 MCT

The one's complement of the contents of the accumulator and L-

.Register replace the previous contents of A and L. All 31 bits of A and L are complemented.

SQUARE

Op-Code 17 SQUARE

K = 0

Square (the contents of A) 3 MCT

The contents of A are multiplied by itself, and the product is stored back into the accumulator and into the L Register. Results are messy if the previous contents of A contain positive or negative overflow.

1.4.7 Assembly Constants

As we saw under Instruction Representation (page 3), the assembly process enables us to change a YUL language instruction into a 15-bit instruction word which will be loaded into memory in binary machine language. At execution time, the hardware fetches the instruction word from memory and sends it to the instruction decoding logic, where it is interpreted and executed.

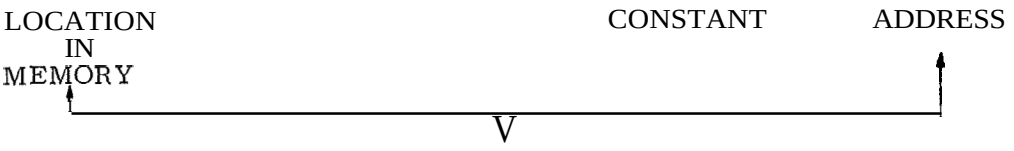
Data (for example: the definition of a constant) is assembled into a 15-bit data word and is loaded into computer memory as a binary number. When this 15-bit data word is fetched from memory, it is treated as a whole. If the programmer has placed the data in his program such that the computer interprets it as an instruction, the program will yield unexpected results.

Within the assembler is a location counter which keeps track of what location we are at in memory. It is important to distinguish between a location in memory, or the address at which a word is located, and the address field within a word, which references some memory address.

Methods exist with which we can create the arithmetic and address constants we wish to use in AGC programs. Those concerning the arithmetic constants will be discussed later, ADRES, REMADR, AND GENADR each create a 12-bit address. FCADR AND ECADR each create a 15-bit constant word containing , respectively, a Fixed Complete Address and an Erasable Complete Address. EBANK = creates an Erasable Bank Declaration (which is not an AGC word) which tells the assembler in which E-Bank the programmer wants subsequent E-Bank addresses to be. BBCON creates a 15-bit Both-Bank-Constant word intended as data to be placed in the BB register (Both Banks Hegister). The last two codes 2BCADR and 2FCADR create Double Complete Addresses including, respectively, a BBCON and an FCADR.

These address constants are necessary for interbank communication. We are able to change Z by setting Z to the contents of a 12-bit address created by a constant, and we are able to specify the E-Bank or F-Bank in which is the location defined in Z through constants which set FB, EB, or BB.

We have said that ADRES, REMADR, and GENADR each create a 12-bit address. (The contents of the three high-order bits of the 15-bit words created will always be equal to zero.) ADRES requires that the current assembly location counter and address values of the ADRES operand be in the same F-Bank or in the same E-Bank.



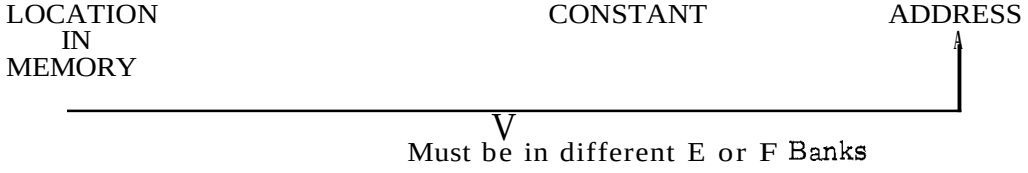
For example, assume that at location 40000_8 in memory we have the constant ADRES 40161_8. Both the location of the word (40000_8) and the address within the word (40161_8) are in F-Bank 14, as required by the constant ADRES. Since the pseudo-address 40161_8 is in fixed-switchable memory, we subtract 10000 from the pseudo-address, getting 30161_8, the augment of 161_8 within F-Bank 14, and we set location 40000_8 to:

0	0	0	0	1	0	0	0	1	1	1	0	0	0	1
15	.	.	.	.	.	.	.	.	.	.	.	.	.	1

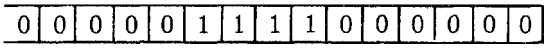
FB Indicators Augment 161 within FB 14

The 01_2 configuration in bits 12 and 11 will cause the FB bits to be appended to the address (at execution time) giving us the address 30161_8.

REMADR requires that the location counter and address values be in different banks.



For example, let us assume that at location 2314_8 in E-memory is the constant word REMADR 1700_8. The location and address values are both in erasable memory but are in different E-Banks, as required by the constant, REMADR 1700_8 forms the 15-bit constant word



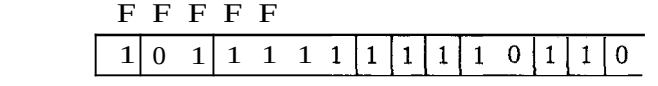
bits 15 10 9 1

300 augment in EB 7

The 11_2 configuration in bits 10 and 9 will signal that the EB bits be appended, giving us the address 1700_8, in EB 7.

GENADR will form a 15-bit.constant word without any checks.

CADR AND FCADR are synonymous codes for the constant which will generate a fixed complete address. The address value within the word must fall in an F-Bank. The 15-bit word generated equals the pseudo-address value minus octal 10000. Bits 15—11 equal the F-Bank number and bits 10—1 equal the relative location of the address in that bank. Let us assume we are in F-Bank 17 and we wish to fetch data from location 67766_8, which is in F-Bank 27. FCADR 67766_8 will create the 15-bit constant word

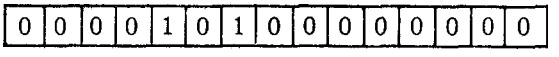


bits 15 11

FB 27 augment 1766_8

which will assist us in switching banks to fetch the data from location 67766_8.

ECADR will create a 15-bit constant word containing an erasable complete address. The address value must be in erasable memory, 0000—3777_8, and the 15-bit word generated equals the 11-bit pseudo-address, Bits 15—12 equal zero. For example, assume we are in FB 4 and wish to get data from location 2400_8 in EB 5. ECADR 2400_8 will create the 15-bit constant word



bits 15 11 9 1

EB 5 Augment 0

EBANK = creates an Erasable Bank Declaration (which is not an AGC word) which tells the assembler that all subsequent references to E-memory must fall within the specified (Operand) E-Bank. The assembler complains whenever an address is equivalent to a location in a different E-Bank. If the EBANK = code is followed by a BBCON, or a 2BCADR, this EBANK = value is good only for one subsequent code, and then the previous EBANK =setting is restored. This is called a

"one-shot EBANK = declaration." Let us assume we have set EBANK = 5, thus informing the assembler that all subsequent E-Bank addresses will be in EB 5. Whenever the assembler hereafter comes upon an address which is not within EB 5, we receive a diagnostic (CUSS). Now suppose that we are in EB 5 and we wish to fetch data from EB 4 once and then return to EB 5. EBANK = 4 enables us to switch (for assembly purposes) from EB 5 to EB 4 for one BBCON word, then to switch back to EB 5 for the remaining subsequent codes. This is the "one-shot EBANK = declaration."

BBCON will create a 15-bit Both-Bank-Constant word intended as data to be placed in the BB register. The address value must be a location in fixed memory (not fixed-fixed) or it must be an F-Bank number (in the range $0-43$). Bits 15—11 of the 15-bit word generated equal the address' bank number. Bits **10—8** and 4 are zeroes. Bits 7—5 are *000* if F-Bank is less than 30, 011 if F-Bank is 30—37, or 100 if F-Bank is 40 43. Bits 3—1 equal the current EBANK = code. Recall that the BB register has the following format:

BB Register

F	F	F	F	F	0	0	0	S	S	S	0	E	E	E

Assume that we have set EBANK = *3*, so that all subsequent E-Bank addresses will be in E-Bank 3. At present, FB 13 is in the Location Counter, and we wish to switch to FB 14. BBCON (FB 14_8) will create the following 15-bit constant word:

0	1	1	0	0	0	0	0	0	0	0	0	0	1	1

bits 15 1

FB 14 EB *3*

2CADR and 2BCADR are synonymous codes which create a Double Complete Address, i. e. a GENADR followed by a BBCON. The code is intended to be used as the operand of a DTCB (DXCH Z) instruction, discussed below. Two 15-bit constant codes are generated by this code. The first word is formed under the rules for GENADR. If the operand address value is in fixed memory, the second word is formed under the rules for BBCON. For an address in erasaale memory, the second word becomes OOOOX where X = the address' octal code EBANK number in the range $0-7$. For example, assume we have set EBANK = 5 so that all subsequent E-BANK addresses will go into EB 5. We are at present in EB **4** and wish to go to location 50000 in FB 20. **2** BCADR will create the following two words:

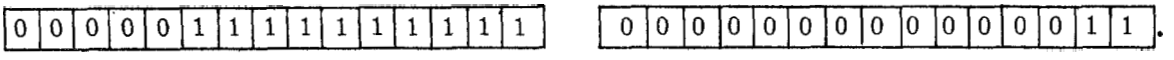
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

1	0	0	0	0	0	0	0	0	0	0	1	0	1

FB **20** EB 5

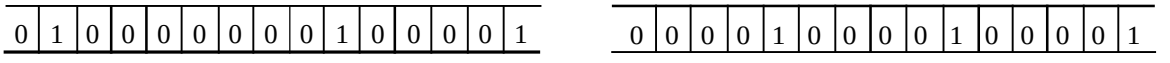
The first word, formed under the rules of GENADR contains the 12-bit address which will become the contents of *Z* when this double precision word is used as an operand under DTCB (DXCH Z). The second word, formed under the rules of BBCON, contains the number of the F Bank which corresponds to the address of the operand. It also contains the number of the E-Bank via our last EBANK = statement.

Now consider that we are in FB 20 and wish to jump to location 337_8 in E-Bank 3. 2BCADR 1777_8 would form the double precision constant word



The first word, formed under the rules for GENADR contains the erasable address which will go into *Z*. The second word contains the address' octal code E-Bank number.

2FCADR creates a Double Complete Address, i. e. an FCADR followed by a GENADR. The address value must be a location in fixed memory, The code is intended as an operand for a DTCF (DXCH FB) instruction, discussed below. This code generates two 15-bit constant words. The first word is formed under the rules for FCADR, and the second is formed under the rules for GENADR. For example, let us assume that we are in EB 3 and wish to jump to location 30041_8 in FB 10_8. 2FCADR 30041_8 would create the following DP word:



FB 10_8 Augment 41

The first word was formed under the rules for FCADR, specifying the F-Bank number of the address in bits 15—11 and the relative location of the address within that bank in bits 10—1. Bits 15—11 of this word become the FB setting when the DP word is used as an operand of a DTCF instruction. The second word, formed under the rules for GENADR, contains the 12-bit address which will be set into *Z* when the DP word is used as an operand of DTCF.

Two implied address codes remain to be discussed,

DTCB

Op-Code 5 DTCB
QC 1
K = 5 (DXCH Z)

Double Transfer Control Switching Both Banks 3 MCT

The contents of the Z and BB registers come into the accumulator and L Register, and the contents of the accumulator and L Register go into the Z and BB registers. For example, suppose that one wants to jump banks for an interrupt. The sequence

```
DCA     OPERAND 1    (Operand 1 is the Address of a 2BCADR)
DTCB
```

would first bring the two constant words created by BCADR (representing Z and BB) into the accumulator and L Register. DTCB would then cause the contents of A and L to go into Z and BB. The present contents of Z and BB would be saved in A and L, and an immediate change of sequence would be in effect.

DTCF

Op-Code 5 DTCF
QC 1
K = 4 (DXCH FB)

Switching F Banks 3 MCT

The contents of FB and Z come into the accumulator and L Register, and the contents of the accumulator and L Register go into FB and Z. A BFCADR is used with this instruction. The first word created by

2FCADR 13177_8

sets FB bits 15—11 to the F-Bank number within which this address is located and sets 10—1 to the relative location of the address within that bank. The second word creates a 12-bit address. The sequence

```
DCA     OPERAND 2  (OPERAND 2 is the address of a 2FCADR)
DTCF
```

will cause the first word created by 2FCADR to come into A, and the Z address in the second word created by BFCADR to come into L. The DTCF will cause the 5-bit FB setting in A to go into FB and the 12-bit address in L to go into Z. The present contents of FB and Z will go into A and L. Thus we have switched F-Banks, sequence control, and have preserved the previous setting of FB and Z in A and L.

List of Assembly Constants

Constant (Op-Code)	Operand (Address)
ADRES	TAG, P. A. (pseudo-address)
REMADR	TAG, P.A.
GENADR	TAG, P.A.
CADR (FCADR)	TAG, P.A.
ECADR	TAG, P.A.
EBANK =	TAG, P.A. , E-BANK #
BBCON	TAG, P. A, F-BANK #
2CADR 2(BCADR)	TAG, P.A.
2FCADR	TAG, P.A.

TAG = Symbolic Tag, e. g. DISRUPTSW
P. A. = Pseudo-Address, e. g. 40273_8
E-BANK # = E-Bank number, *e. g.* 3
F-BANK # = F-BANK number, e. g. 27

1.4.8 Counters

Counters are addressable registers in erasable memory which may be incremented or decremented by special unprogrammed sequences. Two adjacent 15-bit time counters comprise the AGC clock, which has accuracy up to 31 days. Other counters, upon overflow, cause an interrupt of the current program, enabling us to periodically accomplish special processing.

The time counters, designated as Scaler 1 and 2 and Time 1—6, are located in memory as follows:

Octal Location	Counter
24	TIME 2
25	TIME 1
26	TIME 3
27	TIME 4
30	TIME 5
31	TIME 6

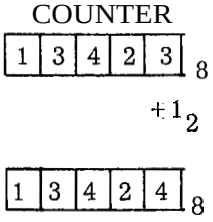
Scaler 1 and Scaler 2 are, respectively, Channels 4 and 3.

The function of counters is to keep track of the state of an external device. For example, the CDU counters monitor the changing state of roll, pitch, and yaw of the spacecraft through the increments and decrements pulsed across the 1/0 interface to the counters by the CDU's. Since the measurement unit of the counter is 40 seconds of arc, the counter will not reflect a change in the position of the spacecraft less than 40 seconds of arc. Increments and decrements to other counters represent different scaling, e. g. milliseconds of time. Counters may be incremented or decremented by making the following requests of the CPU. We describe these hardware sequences by mnemonics not meant to be interpreted as instructions.

PINC

Plus Increment — 1 MCT

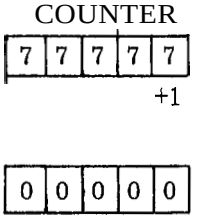
+ 1 is added to the low-order bit of the counter. If the addition results in overflow of the counter, the counter is reset to positive zero.



PCDU

Plus Increment (CDU) — 1 MCT

+ 1 is added to the low-order bit of the counter in two's complement modular (unsigned) notation.



The CDU counters **32 — 36** are incremented and decremented in two's complement notation and are non-algebraic for the hardware sequences PCDU and MCDU. That is, while PINC and MINC would reset the counters to positive or negative zero upon overflow of the counter, PCDU and MCDU increment and decrement the counters in unsigned notation *so* that the quantity 40000_8 would not represent overflow. Since the CDU's are modulo 180^o, we merely continue counting past POSMAX when we reach 180^o (40000_8).

In order to use the readings of the CDU's in one's complement arithmetic calculations, we must convert the two's complement quanties in the CDU counters to signed one's complement notation, Our method of conversion is based on the following considerations. We designate positive zero as the beginning of a revolution and -180^{o} as the midpoint of a revolution. Since the low-order bit of a CDU counter is equal to $\overline{40}''$, $+180^{o}$ $-\overline{40}''$ (37777_{8}) is the closest positive representation of the mid-point that we can have, Similarly, negative zero must be represented as 360^{o} $-\overline{40}''$ (77777_{8}). Thus the difference between $+180^{o}$ and -180^{o} and between +0 and -0 is represented by $\overline{40}''$, which is equal to an increment of 1 to the low-order bit of the CDU counters, This difference of $\overline{40}''$ is equal to the difference between one's and two's complement arithmetic. Therefore, when we read a negative two's complement quantity out of a CDU counter in order to perform one's complement arithmetic with it, we subtract one from the quantity, We need to do nothing to positive quantities since a positive number is identical in one's and two's complement arithmetic. At the end of the calculations, we add one to a negative one's complement number to reconvert it to two's complement notation. As above, we have no need to change a positive one's complement number. An extracode instruction MSU (Modular Subtract) accomplishes the above by differencing two quantities in two's complement notation and leaving the difference in one's complement form,

MINC

Minus Increment — 1 MCT

-1 is added to the low-order bit of the counter. If addition results in overflow of the counter, the contents of the counter are reset to negative zero.

COUNTER

6	7	4	3	1

$_{8}$

-1_{2} MINC: -1 pulsed into bit 1

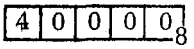
6	7	4	3	0

MCDU

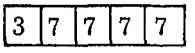
Minus Increment (CDU)

- 1 is added to the low-order bit in two's complement modular (unsigned) notation.

COUNTER



MCDU: -1 pulsed into bit 1 in unsigned notation



See remarks under PCDU

DINC

Diminishing Increment 1 MCT

If the contents of the counter are greater than positive zero, the contents are decremented by + 1.

If the contents of the counter are less than negative zero, the contents of the counter are incremented by +1.

If the contents of the counter are equal to positive or negative zero, the contents are left unchanged.

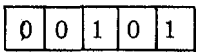
In other words, we move toward zero from either a positive or negative direction.

SHINC

Shift Increment 1 MCT

The contents of the counter are shifted left one bit. If positive overflow results, an interrupt request will be set for the counter.

COUNTER

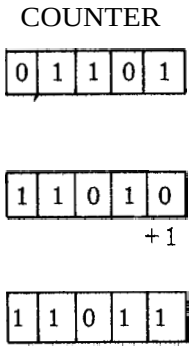


Contents of counter shifted left 1 bit

SHANC

Shift and Add Increment 1 MCT

The contents of the counter are shifted left one bit and + 1 is added to the low-order bit of the counter. If positive overflow of the counter results, an interrupt request will be set for the counter.



SHINC and SHANC are used for serial to parallel conversion (converting a stream of bits coming in bit by bit *so* that it may be accessed as a whole word).

These unprogrammed sequences may occur only at the end of an instruction sequence. An instruction sequence may be only one instruction such as TC K or ADS K, or it may be a series of instructions such as

```
EXTEND
INDEX     K
DCA       K.
```

If a PINC, MINC, etc. request is made while an instruction sequence is still being processed, it will wait in a circuit until the instruction sequence processing is completed. The request will then be serviced. These unprogrammed sequences causing counters to increment and decrement take place between instruction sequences *so* that no counter will be in danger of changing while a sampling of the counter is being taken by the instruction processing.

The counters vary in the type of overflow processing which they cause. The contents of some counters are reset to zero upon overflow; in other cases, the overflow is lost. More frequently, overflow of a counter causes an interrupt.

There are many types of counters other than timers. A list and partial description of the counters in the AGC is given in Chart I on Page 64.

1.5 Interrupt Processing

The normal sequence of instruction processing for the current program can be interrupted for special processing through RUPT's. The two main functions of RUPT's are to allow automatic monitoring and to allow control over intervals of time (AT).

Concerning automatic monitoring, it is often necessary for the system to respond immediately to some external signal or situation. In the absence of interrupts, we would have to require all programs to frequently monitor such signals and situations. To the programmer, this could be easily a very burdensome task. Instead, by having direct communication between external signals/conditions and hardware interrupts, external events can automatically lead to processing by some central program. We thus guarantee that the system will react instantly to certain external signals and conditions, and we remove a programmer burden.

Concerning control over intervals of time, we may assume that it will be necessary for a program to wait an interval of time before it resumes processing. We may also assume that there exist some system functions which regularly (every AT) must be serviced. By connecting time counter overflows to hardware interrupts, we can preset the counters *so* that after AT they will yield special processing, such as returning to a program that wished to wait **AT,** or returning to servicing some regular system function,

When a RUPT has caused a transfer of control from the main program to a prespecified RUPT location, the states of the central registers A, (Z, B,) Q, and BB may (will) be preserved, if desired, in temporary storage. Upon completion of the RUPT sequence, a RESUME instruction restores the central registers to their previous states and returns control to the previously interrupted program.

An interrupt cannot occur under the following conditions:

1. while a RUPT is currently being processed;
2. while there is overflow in A;
3. while the extracode switch is on (implying that the instruction sequence has not been processed to completion);
4. if the INHINT command has been given without a subsequent RELINT.

Condition 1 may be repealed by completing a RUPT processing and by giving the command RESUME. The INHINT command in condition 4 may be rescinded by commanding RELINT.

1.5.1 The Clock and Scalar

As we have said above, the AGC clock is composed of two 15-bit adjacent counters in memory, called TIME 1 and TIME 2, which can keep time for 31 days. Time 1 is scaled to be accurate to 10 ms. That is, 1 centisecond or 10 ms must elapse before the low-order bit of TIME 1 can be incremented by 1 (PINC'd).

For greater timing accuracy, we can access SCALER 1 (14 bits long) and SCALER 2 (the time counters in channel 4 and 3, respectively). The low-order bit of SCALER 1 is incremented by 1 every 1/1600th of a second, A pulse into bit 5 of SCALER 1 not only increments bit 5 but PINC's bit 1 of TIME 1. Thus is TIME 1 incremented every centisecond, or 10 ms. The overflow from bit 14 of SCALER 1 increments bit 1 of SCALER 2, SCALER 2 is thus incremented every 10.24 seconds. Together, the scalers can keep time for 23.3 hours. Overflow from bit 14 of TIME 1 PINC's bit 1 of TIME 2 and resets TIME 1 to + 0. Overflow from bit 14 of TIME 2 is lost. Thus the scalers and TIME 1 and 2 can monitor time up to 31 days.

Time counters T3 and T4 are incremented in the same manner as T1, but overflow from bit 14 of these counters triggers an interrupt. Thus, to cause an interrupt AT from now, as is often necessary, we set a timer such as TIME3 equal to its maximum, plus one (or 1.0) minus AT. An interrupt will occur within AT-X from now with X less than 10 ms. This merely means that T3 may get its first 10 ms increment before a full 10 ms has elapsed because the pulsing of the timers is asynchronous to instruction execution time. Time 3 and TIME 4 are phased to be 5 ms apart in pulsing. As long as RUPT processing does not exceed 4 ms, they will not interfere with each other.

Let us consider a T4RUPT as an example of a programmed interrupt. The following description may be followed in the diagram of Programmed Interrupts on page 1-69. Overflow from bit 14 of TIME 4 generates a pulse which will set the RUPT indicator bit for T4RUPT. The hardware, scanning the RUPT indicator bits will service the RUPT of highest priority whose indicator bit is set, by closing its RUPT switch. If the INHINT command has been given in the main program, the pulse will wait at the open INHINT-RELINT switch until the switch is closed by a RELINT command. When the INHINT-RELINT switch closes, the pulse continues to the final

switch, The switch is open and the pulse will wait if the extracode bit is on (implying that an instruction sequence in the main program has not yet been processed to completion), or if overflow exists in A, or if a RUPT is already in progress, When any of these conditions is removed (e.g. a RESUME command closes the switch), there is no further switch to stop the pulse from triggering the special processing of the T4RUPT.

As in all interrupts, the hardware now causes the contents of the Z Register to be saved in a temporary storage register ZRUPT at location 15_8 and the contents of the B Register, containing the next instruction, to be saved in the temporary storage register BRUPT at location 17_8. The hardware now transfers control from the main program to the location at which the processing caused by a T4RUPT begins. At this new location, program processing ordinarily will save the contents of A, L, Q, and BB in temporary storage registers ARUPT (10_8), LRUPT (11_8), QRUPT (12_8), and BBRUPT (16_8), and procede to fulfill whatever functions are required of this particular RUPT. Afterwards, the program restores A-L-Q-BBRUPT's to A, L, Q, and BB. The instruction RESUME causes the hardware to restore Z from ZRUPT and select the contents of BRUPT as the next instruction. Thus, a program which was once interrupted for T4RUPT processing may now continue as if nothing has happened,

CHART I

Summary of Counters for CSM & LEM

Note: ( ) Parenthesis indicates use of counter on LEM as different for CSM.

Octal Location	Symbol	Name	Scaling	Use
24	TIME 2	T 2	10 MS	Elapsed Time
25	TIME 1	T 1	10 MS	Elapsed Time
26	TIME 3	T 3	10 MS	Wait-List
27	TIME 4	T 4	10 MS	T4RUPT
30	TIME 5	T 5	10 MS	Digital Auto Pilot
31	TIME 6	T 6	1/1600 Sec.	Fine Time for Clocking
32 - 34	CDUX, Y, Z	Inner, Middle, Outer-Gimbals	40" Arc	Relate Stable Member Axis to Body Axis
35	OPTY	Optics Trunnion {or Radar)	10" (or 40") Arc	Relate Line of Sight to Body Axis.
36	OPTX	Optics Shaft (or Radar)	40" Arc	
37 - 41	PIPA, X, Y, Z	X, Y, Z - Stable Member	5.85 CM/Sec. (or 1 CM/Sec.)	Measure Change in Velocity
42 - 44	Spare in CSM (or RHCP, Y, R)	(Rotational Hand Controller Inputs for Pitch, Yaw, Roll)	(1 up to ± 31)	(Manually Command an Attitude Roll, Pitch, Yaw).
45	INLINK	Uplink		Up-Telemetry

CHART I continued

Octal Location	Symbol	Name	Scaling	Use
46	RNRAD	Rendezvous & Landing Radar Data		Parallel to Serial Conversion
		RR Range Low	+9.38 ft.	
		RR Range Low	8 x 9.38 ft.	
		LR VX High	+0.6435 ft./sec.	
		RR Range	+0.6278 ft./sec.	
		LR VY	+1.2525 ft./sec.	
		LR VZ	+0.8571 ft./sec.	
		LR Altitude, Low	+1.079 ft.	
		LR Altitude, High	+4.9977 x 1.079 ft.	
47	Gyro CTR	Out Counter for Gyros	2ss Rad. 22 1	Drift Compensation and Fine-Align the Platform
50 - 52	CDUYCMD X Z	Outcounters for CDUs		Used for Changing the DAC Error Counter in CDU
53	OPTYCMD	Outcounter for Optics (or Radar)	40" (or 160")	Drives Optics (or Rendezvous Radar) or is used by Digital Autopilot
54	OPTXCMD	Outcounter for Optics (or Radar)	160" Arc	
55 - 56	Spare			
57	OutLink	Cross-Link		Parallel to Serial LEM and CSM Telemetry
60	(ALTM)	(Altitude Meter)	2.345'	(Drives Inertial Data Display for Altitude on LEM)

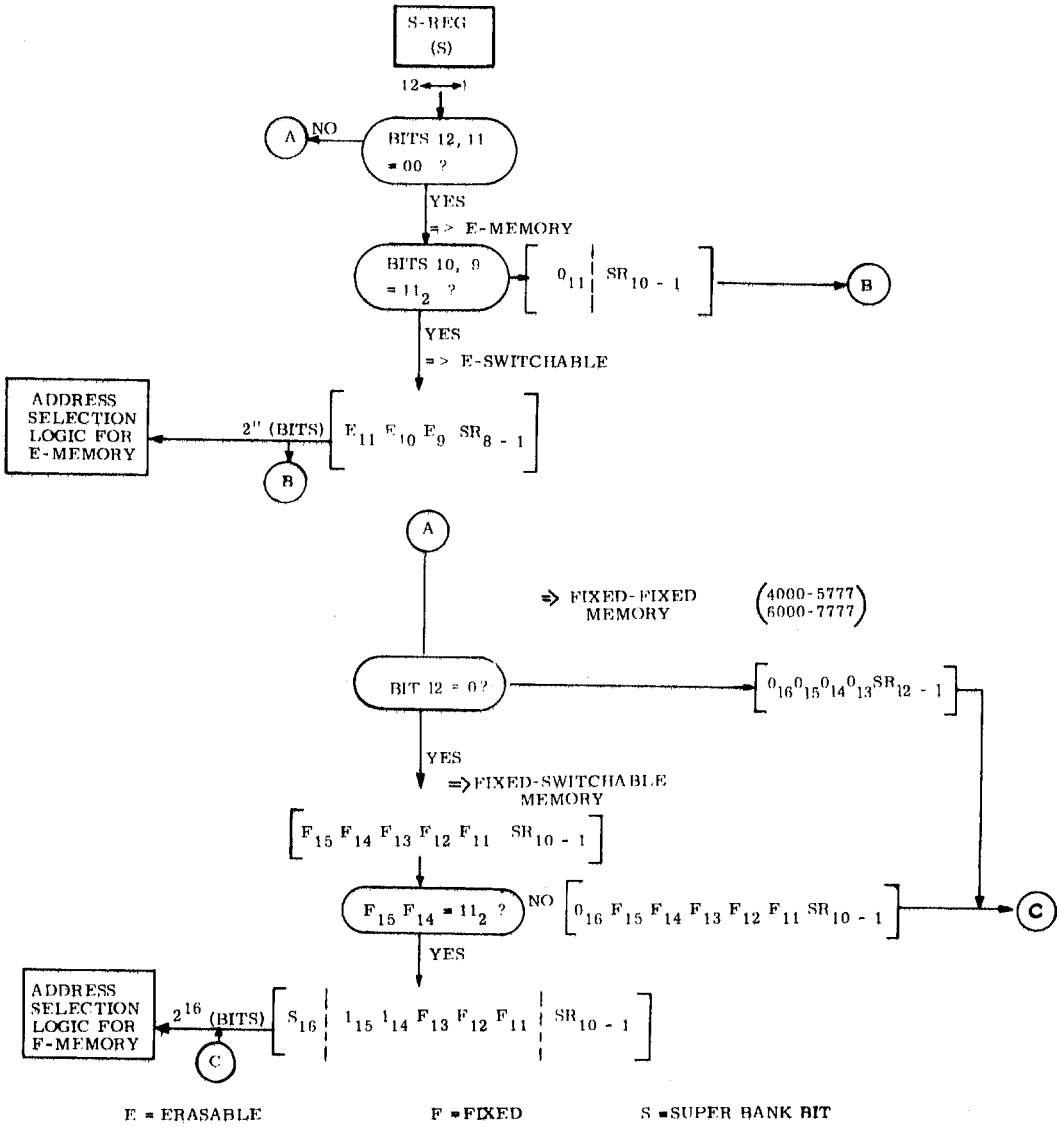
CHART 2
MEMORY LAYOUT

F BANK	NAME	SUBTRACT	BANK SIZE	PSEUDO ADDRESS	3-BIT BANK
x	Erasable	0	256	0-377	0
x	Erasable	0		400-777	1
x	Erasable	0		1000-1377	2
x	Erasable Switched	1400 & Append		1400-1777	3
	Erasable Switched	F Bank		2000-2377	4
	Erasable Switched	Hits 11, 10, 9 in Erasable Memory		2400-2777	5
	Erasable Switrhed	Selection Logic		3000-3377	6
	Erasable Switched			3400-3777	7
02	Fixed/Fixed	0 & Use 12-Rit Address in	1024	4000-5777	x
03	Fixed/Fixed	Address Selection	1024	6000-7777	
00	Fixed	2000 & Append F_{15} - F_{11} in Fixed	1024	10000-11777	
01	Fixed	Address Selection	1024	12000-13177	
02	Redundant Because	Non-Existent Address		14000-15777	
03	Used Above			16000-17777	
04	Fixed	2000	1024	20000-21777	
05	Fixed				
	Fixed				
07	Fixed			26000-27777	
10-13	Fixed			30000-37777	
14-17	Fixed			40000-47777	
20-23	Fixed			50000-57777	
24-27	Fired			60000-67777	
30-33	S = 0 Super Bank 0			70000-77777	
30-33	S = 1 Super Hank 1			110000-117777	
34-37	S = 0			100000-107777	

36 Fixed Ranks and 2 Erasable Ranks or 36864 + 2048 = 38912 locations

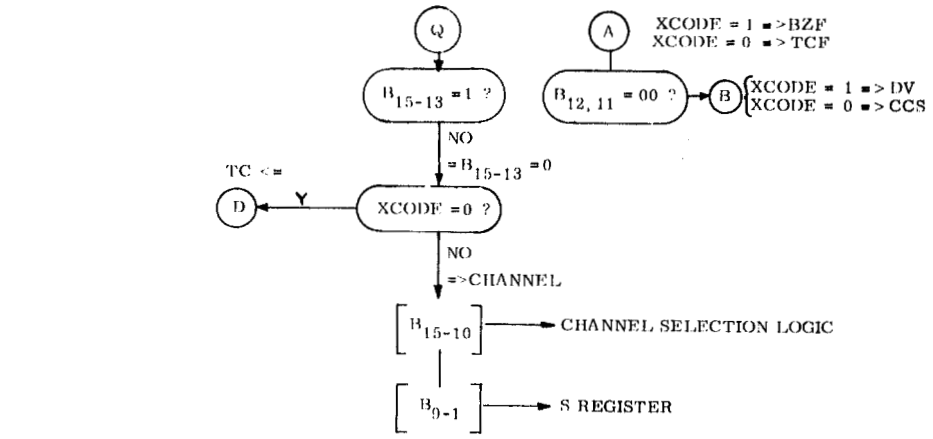
,CHART 3

ADDRESS SELECTION LOGIC



E = ERASABLE F = FIXED S = SUPER BANK BIT

CHART 3 continued



F = ERASABLE F = FIXED B = B-REG S = S-REG

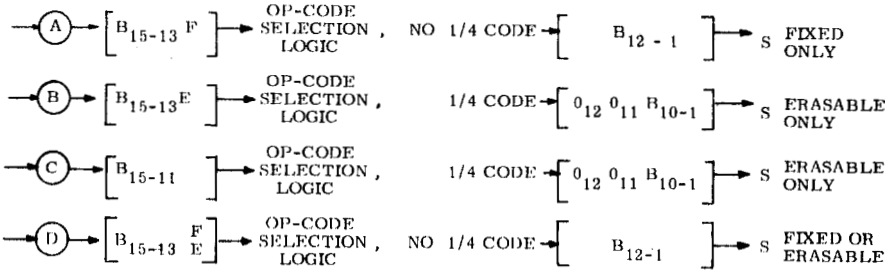
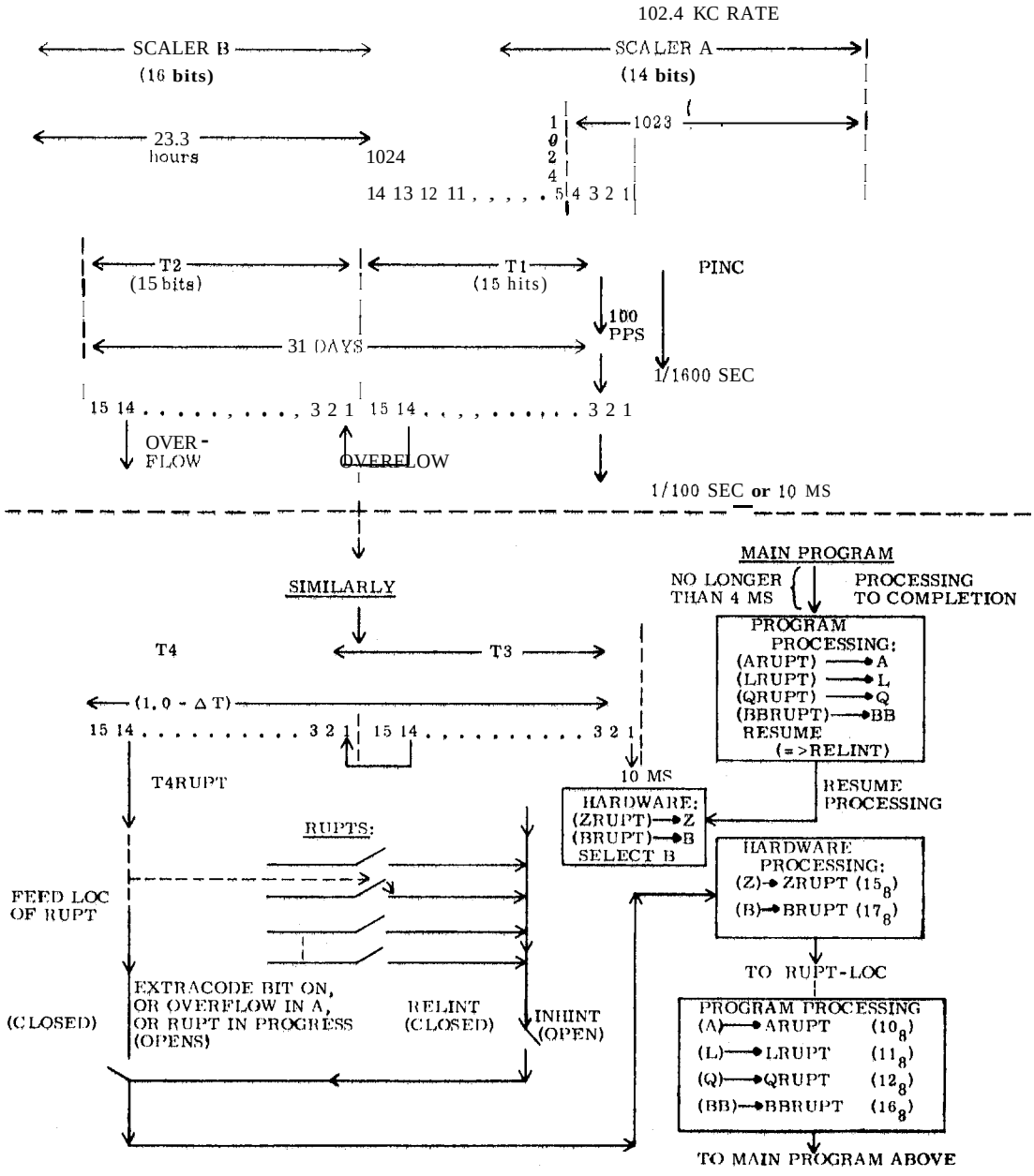


CHART 4

PROGRAMMED INTERRUPTS



2. THE INTERPRETER

2.1 Introduction

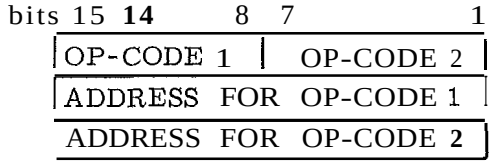
The Apollo Guidance Computer was designed with the idea that its weight, size, and power supply were costly items. Mission requirements warrant a hardware compromise of a word-length with a minimum of 15 bits and an instruction repertoire of 33 instructions with which to work, The result, therefore, is a small, fairly simple machine with limited abilities. While the AGC hardware provided for manipulation of single- and double-precision quantities, frequent need arose to handle multi-precision quantities, trigonometric operations, vector and matrix operations, and extensive scalar operations, Thus, to fulfill the system requirements planned for the lunar missions within the constraints of hardware limitations, it is necessary to employ software to expand the capabilities of the AGC.

One method of accomplishing this would be through a collection of subroutines. By creating within the computer a large library of subroutines which perform various higher level arithmetic and language operations, we could save mission programmers the burden of having to code their complicated operations in extensive sequences of basic machine instructions, This approach has two disadvantages, however. First, since programmers would be calling subroutines often, a great deal of memory would be taken up merely with the frequently repeated calling sequences, Secondly, much of memory would be taken up as temporary storage for the contents of registers which programmers needed for later processing. For example, to call the subroutine *XYZ* which requires two arguments,

```
TC      XYZ
CADR    ARG 1
CADR    ARG 2
```

three words of memory are used just in calling the subroutine while additional words are used for temporary storage.

Thus, to solve the memory wastage problem caused by frequent use of the calling sequences, it is expedient to create an entirely special mnemonic language in which each mnemonic corresponds to a subroutine. Since, in many cases, the new mnemonic instructions require no addresses, we design a packed instruction format which stores two seven-bit operation codes in one word of memory and any required address constants in the two following words:



To interpret our special mnemonic language, we design a central subroutine which will encode the instruction formats (and use common temporaries) and execute the required subroutine sequence of AGC instructions.

Each subroutine is constructed such that the combination of single-operation AGC instructions forms a particular method of doing some higher level operation (such as obtaining a square root) required frequently by mission programmers. Thus, programmers have access to procedure-oriented operations without having to learn various subroutine-calling sequences. We similarly aid engineer-programmers by naming the mnemonics with the vocabulary oriented to their specialized work.

By building a "software" machine with the kind of programming designs discussed above, we achieve, for programming purposes, a larger, more diversified computer than the basic AGC. In order to "build" our software machine, we need to create the components which will simulate their hardware counterparts.

MPAC We design a multi-purpose accumulator with seven 15-bit registers so that multi-precision quantities may be easily manipulated. The three types of quantities which may be contained in the accumulator are (1) a double-precision quantity occupying the pair of registers MPAC and MPAC + 1 with magnitudes up to $1-2^{-28}$; (2) a triple-precision quantity occupying the three registers MPAC, MPAC + 1, and MPAC + 2 with magnitudes up to $1-2^{-42}$; and (3) a column vector quantity occupying the six registers MPAC, MPAC + 1, MPAC + 3, MPAC + 4, MPAC + 5, and MPAC + 6 representing an X, Y, Z, DP vector.

OVFIND An overflow indicator functions similarly to its AGC hardware counterpart in recording for the current program the fact that an instruction operation has created overflow. Just as we could test for overflow in an AGC program with an OVSK instruction, *so* we use the instructions BOV (Branch on Overflow) and BOVB (Branch on Overflow to Basic) to test for overflow in interpretive programs.

ADRLOC The address location register is the interpretive counterpart of the basic register *Z*. It is the program counter which contains the next address in memory from which an interpretive instruction will be taken.

QPRET We need a return address register to serve as the counterpart of the AGC Q-Register in preserving the location at which we shall resume processing the main program when we return from a subroutine, Just as TC left in Q the complete address of the next AGC instruction, so the interpretive instruction CALL leaves in QPRET the complete address of the next interpretive instruction.

X1 and X2 In case a programmer wishes to modify the address portion of instructions through the use of index registers, we provide two for the purpose. If an address is indexed, the contents of the specified index register are subtracted from the unmodified address, yielding the net operand address,

S1 and S2 The two step registers may be used as temporary storage for single- or double-precision quantities, but are designed principally to decrement X1 and X2 in loops.

PUSHLOC We design the push-down location register to function as a location pointer for the push-down list just as *Z* functions as the program counter for AGC instructions and ADRLOC for interpretive instructions.

PUSH-DOWN LIST We create a "push-down" list as a means of saving memory by using implied address schemes to specify temporary storage, and as a means of providing the convenience of having our machine temporarily store quantities without programmer intervention. The list may contain 38 15-bit quantities with the characteristic that the last quantity to be entered (pushed down) is the first quantity to be withdrawn (pushed up).

Besides these registers, we need to design instruction formats to accommodate multi-precision scalar arithmetic, trigonometric operations, and vector and matrix operations. Since many of our vector and scalar instructions require no address, we design the packed format for general instructions discussed above.

We represent memory in three groups rather than in banks. Local erasable memory corresponds to the general erasable locations 61_8 to 1377_8 plus the current E-bank. * The low half-memory corresponds to fixed memory banks 4 – 17 and the high half-memory consists of fixed memory banks 21 – 37. **

To represent data, we create formats for single-precision quantities, double-precision quantities, column vectors, and matrices.

Lastly, we must of course design the software equivalent of a central processing unit to perform the functions of encoding instructions, creating effective operand addresses, and executing the instructions.

The Assembler creates packed-format mnemonics for interpretive instructions in the same manner it translates basic machine language into executable code. Reading an interpretive instruction, the Assembler transforms the first mnemonic operation code to a 7-bit op-code which it packs into the left hand operand field, It packs the 7-bit translation of the second op-code into the right hand address field. If both op-codes take defined addresses, the Assembler transforms the address of the first op-code to a 15-bit address field directly below the second op-code. The Assembler translation of the address of the second op-code is placed directly below the first address.

OP-CODE 1	OP-CODE 2
	ADR 1
	ADR 2
OP-CODE 3	OP-CODE 4
	ADR 3
	ADR 4
OP-CODE 5	etc.

* Soon to be changed such that local E-memory includes all the erasable memory.
** Soon to be changed to banks 22 – 37.

If an instruction contains only one operation code and its address, it is coded thus:

```
OP-CODE 1
                    ADR 1
```

At execution time, any address found which does not have an operation code is considered to be an address in which data will be stored. Thus, the STORE operation code and the address appear on the same line:

```
STORE               STORADR
```

Unfortunately, confusion results when OP-CODE 1 takes an address, OP-CODE 2 pushes up, and a store operation is the next instruction. At execution time, this sequence would appear thus:

```
OP-CODE 1           OP-CODE 2
                    ADR 1
                    STORADR
```

Since STORADR would be considered the address of OP-CODE 2, the Assembler requires that the STORADR address be preceded by a STADR code.

```
OP-CODE 1           OP-CODE 2
                    ADR 1

STORE               STORADR
```

Thus, the STORADR is not processed as the operand address of OP-CODE 2.

The other situation requiring the use of a STADR code is:

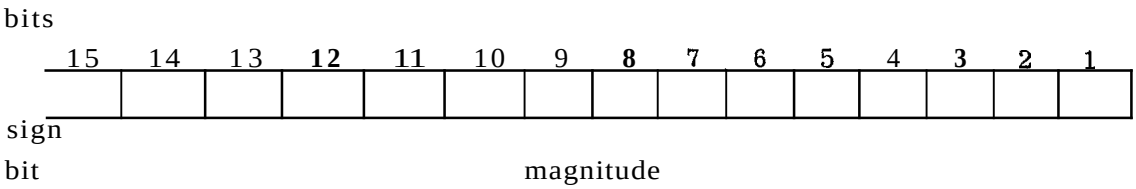
```
OP-CODE 1           STADR

STORE               STORADR
```

This concludes the introduction to the interpreter. Our discussion will treat the interpreter as a machine. Bear in mind that we are really describing a program.

2.2 Memory

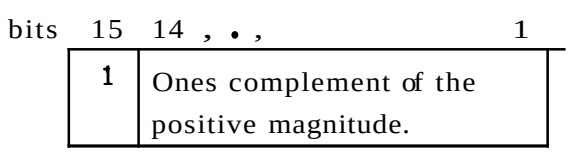
A word in interpretive language is composed of 15 binary bits, numbered from left to right as bit 15, 14, . . ., 1. Bits 14–1 contain the magnitude of a quantity and bit 15, the sign of the quantity. A sixteenth "parity" bit exists solely for internally verifying that the hardware is functioning normally.



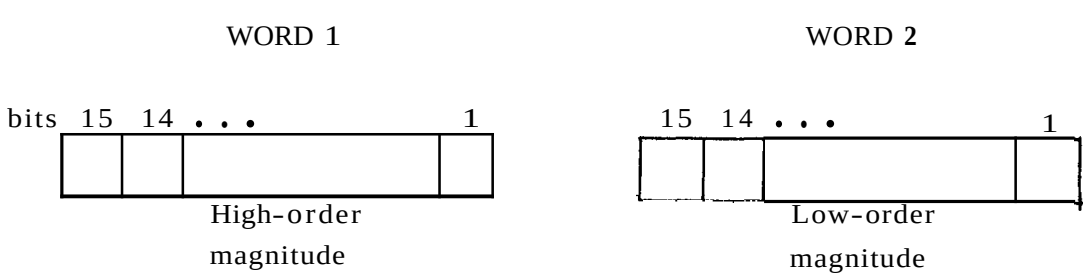
2.2.1 Data Representation

We allow data to be represented as signed, fractional, single-precision quantities, double- and triple-precision quantities, column vectors, and matrices. The arithmetic is fixed-point throughout, with the binary point falling between bits 15 and 14.

Thus one word, which forms a Single Precision (SP) quantity, has magnitudes up to $1-2^{-14}$. If bit 15 contains a one, then bits 14–1 are the ones complement representation of the positive magnitude.

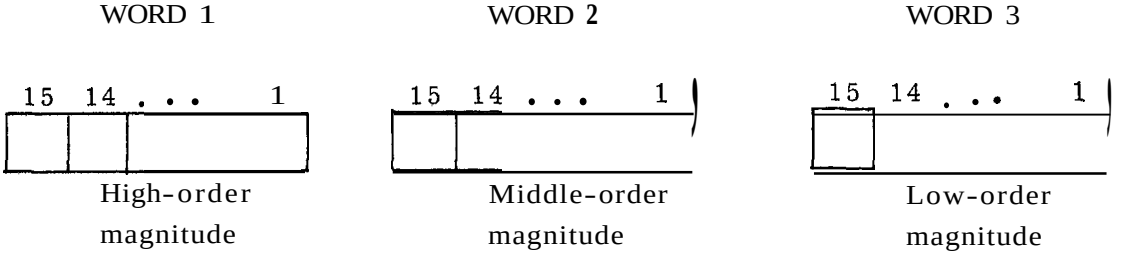


Since we frequently require precision beyond fourteen magnitude bits, a Double Precision (DP) quantity, which consists of two adjacent words, provides us with magnitudes up to $1-2^{-28}$. Although the sign in bit 15 of the second word may occasionally differ from the sign of the first word, the sign of the DP quantity is understood, usually, to be the sign in bit 15 of the first word. Bits 14–1 of the first word contain the high-order magnitude bits of the quantity, while bits 14–1 of the second word contain the low-order magnitude bits.

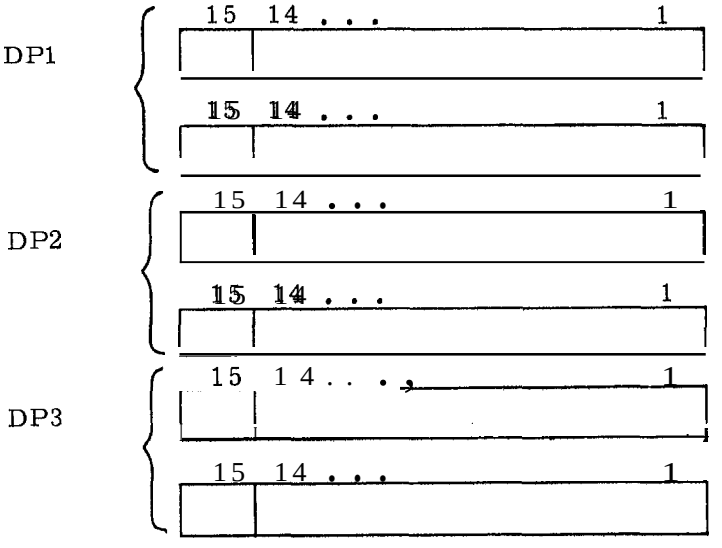


As in Single-Precision words, if a quantity is negative (i.e., bit 15 = 1) then the magnitude bits represent the ones complement form of the quantity's positive magnitude,

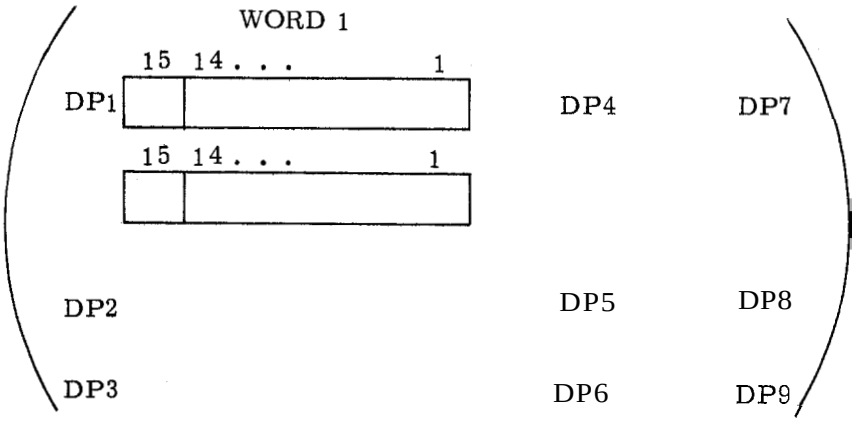
For greater precision, we provide a Triple-Precision (TP) format which allows quantities to be defined within 3 words with magnitudes up to $1-2^{-42}$. As above, all 42 magnitude bits exist in complemented notation if the sign is negative.



A Vector quantity may be represented in six words as three Double-Precision quantities:

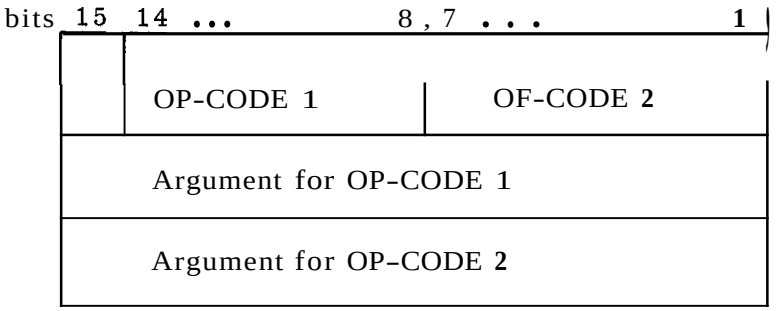


A Matrix quantity may be represented within eighteen words as nine Double-Precision quantities.

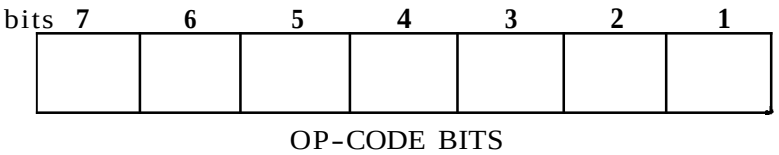


2.2.2 Instruction Representation

Since many of the interpretive instructions take no specified arguments, we save memory by packing two operation codes in one word and any required arguments in the two following words:



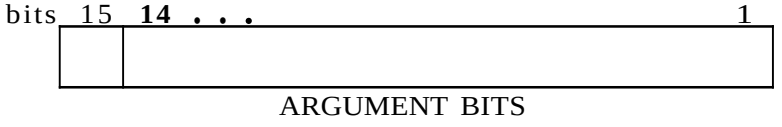
Seven bits are provided to define $2^7-1 = 127$ different interpretive operations.



These 127 operations are broadly divided into four classes according to the configuration in op-code bits 2 and 1. Bit 1 is considered to be the address bit, with a 1 configuration implying that the operation takes an argument. Bit 2 may be thought of as the index bit, with a 1 configuration indicating that the instruction may be indexed.

A 00 configuration in bits 2 and 1 of an op-code implies, therefore, that the instruction does not take an argument and may not be indexed. Such unary instructions include the scalar functions for obtaining a square root, cosine, sine, etc.

Conversely, a 11_2 configuration in bits 2 and 1 of the op-code indicates that the instruction takes an address and may be indexed. Bit 15 of the argument is used to specify which index register, X1 or X2, will be utilized.

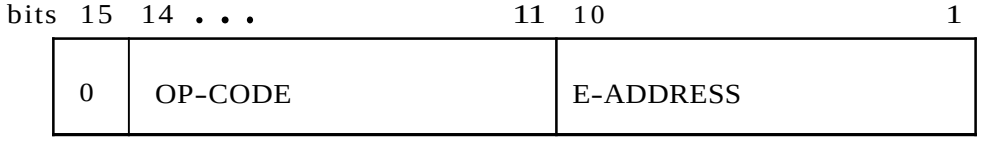


A 0 in bit 15 implies that index register 1 (X1) is to be used; a 1 implies that X2 will be used. The Assembler will of course treat an argument with a 1 in bit 15 as a negative quantity and will represent it in its ones complement form. Thus, if X1 is specified, then the argument is a quantity equal to or greater than 0, and if X2 is specified, the argument is negative. (In fact, it is the positive address + 1, complemented.)

A 01 configuration in bits 2 and 1 of the operation code implies that the instruction takes an argument but may not be indexed. Since we therefore do not need bit 15 of the argument to specify an index register, we use it to distinguish between having a specified address and requiring an address from the Push-down List, This is accomplished by giving all operation-code words the characteristic of having a 1 in bit 15. We then set bit 15 of the arguments whose op-codes fall in class 01 equal to 0. Thus, if the Interpreter does not find the specified operand address with bit 15 = 1, it will encounter the next operation code with bit 15 = 1 and will know that since the op-code required an argument, it must fetch the argument from the Push-down List. Arguments in the Push-down List may not be indexed, since we require the use of bit 15 just to specify the Push-down List and it cannot conflict with using bit 15 to specify X2.

Since we have exhausted the configurations of 00, 11_2, and 01 to indicate general classes of operations, we group all others under the remaining configuration of 10_2. These are the branching instructions and the index instructions which modify the contents of the index registers, We may not index an index instruction since we need the use of bit 15 of the argument to specify which index register is involved. Furthermore, as is true of branching instructions as well, we have no way of indicating any desired indexing as bit 2 of the op-code for this class is always 1 to signal the class of BRANCH/INDEX instructions.

Since the addresses of store operations must be located in erasable memory, we create a special format which packs a 4-bit store operation code and its 10-bit erasable address into one word:



Thus we may reference erasable memory through location 1777_8* (2^{10}-1).

We categorize interpretive instructions into the following eight more specific instruction groups:

1) Memory Load and/or Store Instructions
   These instructions transfer data to and from storage locations.

2) Control Instructions
   This group effects sequence changes of instructions.

3) Decision Instructions
   These instructions test the results of arithmetic operations,

4) Switch Instructions
   This group manipulates and tests the switches.

---

* A store instruction format is currently being implemented which will provide for an 11-bit address portion, thus rendering accessible all of erasable memory. Since this 11th bit is being taken from the op-code field, the op-code portion will consist only of bits 14, 13, and 12. Because bit 11 has been used to indicate the indexable characteristic of some store op-codes, its loss results in loss of the ability to index the first operand of STODL and STOVL instructions, and to index both operands of STODL and STOVL instructions at the same time.

5) Index Register Instructions
These instructions manipulate and test the index register.

6) MPAC Instructions
These instructions manipulate data in the MPAC without affecting memory.

7) Arithmetic Instructions
These instructions perform arithmetic operations with both memory and the MPAC.

8) Miscellaneous Instructions
Instructions which do not fall into any of the previous categories.

The characteristics of these instruction categories are shown in the charts starting on page 12. For each instruction, the following information is provided:

1) Whether the instruction takes 1, 2, or no operands;

2) The nature of the operands and how they may be modified; and

3) Significant side effects of the instruction,

This information is discussed under the two headings OPERAND-1 and OPERAND-2. Under each heading are the columns entitled A/C, P, *, i, E, and F. An instruction takes no operands if column A/C under OPERAND-1 is blank. An "A" in the column indicates that the instruction refers to the contents of an address. A "C" indicates that the instruction uses the numerical value (i.e., is a constant) of the address, A check under any of the following columns indicates that the argument may or must reference the Push-down List (P), be indexed (*), may indirectly address (i), refer to erasable memory (E), or refer to fixed memory (F)—both references are possible. If column A/C under OPERAND-2 is blank, the instruction takes 1 operand address at most. Otherwise, the instruction takes two operands.

Certain side effects of the instructions are recorded by the columns **MPAC,** OVFIND, ABORT, and SEE. A check under MPAC denotes that the instruction may alter the contents of MPAC. Checks under OVFIND and ABORT have similar meanings. An "R" under OVFIND indicates that this instruction resets the overflow indicator (OVFIND).

OP-CODE	Address/ Constant	Push/Down	Index	Indirect	Memory		Address/ Constant	Push/Down	Index	Indirect	Memory					
	OPERAND - 1						OPERAND - 2									
	A/C	P	*	i	E	F	A/C	P	*	i	E	F	MPAC	OVFIND	ABORT	SEE
MEMORY LOAD/STORE																
STORE	A		√		∨											
DLOAD	A	√	√		√	√							√			
TLOAD	A	√	√		√	√							√			
VLOAD	A	√	√		√	√							√			
SLOAD	A		√		√	√							√			
STODL	A				√		A	√	√		√	√	√			PDDL
STOVL	A				√		A	√	√		√	√	√			PDVL
STCALL	A			√	√		A			√	√	√				
STQ	A				√											
CONTROL																SWITCHES
GOTO	A			√	∨	∨										
CGOTO	A		√		√		C					√				
CALL	A			√	√	√										
CCALL	A		√		√		C					√				
RVQ				√	√											QPRET
RTB	A					√										
EXIT																

OP-CODE	OPERAND - 1						OPERAND - 2									
	Address/ Constant	Push/Down	Index	Indirect	Memory		Address/ Constant	Push/Down	Index	Indirect	Memory					
	A/C	P	*	i	E	F	A/C	P	*	i	E	F	MPAC	OVFIND	ABORT	SEE
DECISION																
BPL	A			√	√	√										
BZE	A			√	√	√										
BMN	A			√	√	√										
BOV	A			√	√	√								R		
BOVB	A					√								R		
BHIZ	A			√	√	√										
SWITCHES (0-119)																
SET	A															
SETGO	A						A			√	√	√				
BONSET	A						A			√	√	√				
BOFSET	A						A			√	√	√				
CLEAR	A															
CLRGO	A						A			√	√	√				
BONCLR	A						A			√	√	√				
BOFCLR	A						A			√	√	√				
INVERT	A															

OP-CODE	OPERAND - 1						OPERAND - 2									
	Address/ Constant	Push/Down	Index	Indirect	Memory		Address/ Constant	Push/Down	Index	Indirect	Memory					
	A/C	P	*	i	E	F	A/C	P	*	i	E	F	MPAC	OVFIND	ABORT	SEE
SWITCHES (0-119) Cont.																
INVGO	A						A			√	√	√				
BONINV	A						A			√	√	√				
BOFINV							A			√	√	√				
							A			√	√	√				
							A			√	√	√				
INDEX REGISTERS																
AXT (1, 2)	C															
AXC (1, 2)	C															
LXA (1, 2)	A				√											
LXC (1, 2)	A				√											
SXA (1, 2)	A				√											
XCHX (1, 2)	A				√											
INCR (1, 2)	C															
XAD (1, 2)	A				√	√										
XSU (1, 2)	A				√	√										
TIX (1, 2) Step Regs	A			√	√	√										

OP-CODE	OPERAND - 1						OPERAND - 2									
	Address / Constant	Push/Down	Index	Indirect	Memory		Address / Constant	Push/Dow	Index	Indirect	Memory					
	A/C	P	*	i	E	F	A/C	P	*	i	E	F	MPAC	OVFIND	ABORT	SEE
MPAC MANIPULATIONS																
PDDL	A	√	√		√	√							√			
PDVL	A	√	√		√	√							√			
SIGN	A				√								√			
SQRT													√		√	
DSQ													√			
VSQ													√	√		
ROUND													√	√		
DCOMP													√			
VCOMP													√			
ABS													√			
ABVAL													√	√		
UNIT													√	√		
VDEF	√												√			
SIN, COS													√			
ASIN, ACOS													√		√	
SL (1-4) (R)													√	√		

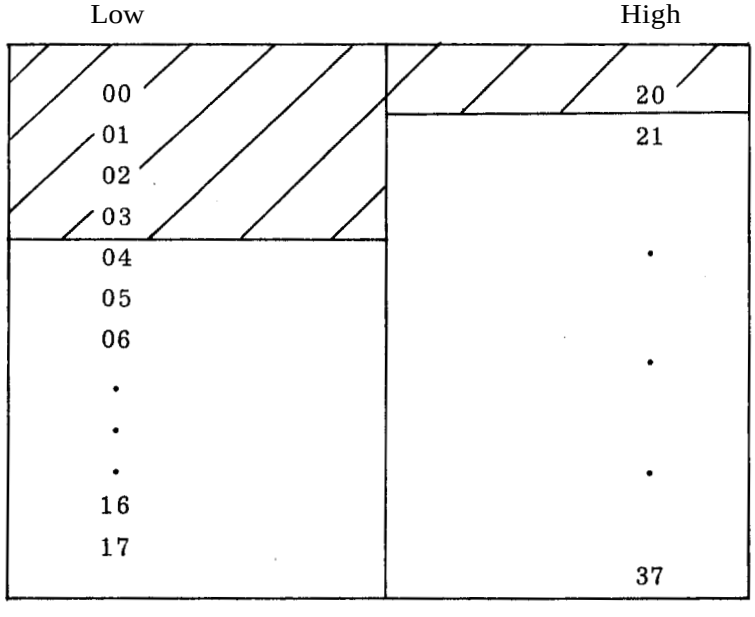
OP-CODE	OPERAND - 1						OPERAND - 2									
	Address/ Constant	Push/Down	Index	Indirect	Memory		Address/ Constant	Push/Down	Index	Indirect	Memory					
	A/C	P	*	i	E	F	A/C	P	*	i	E	F	MPAC	OVFIND	ABORT	SEE
MPAC MANIPULATIONS Cont.																
SR (1-4) (R)													√			
VSL (1-8)													√	√		
VSR (1-8)													√			
SL (R)	C		√										√	√		
SR (R)	C		√										√			
VSL	C		√										√	√		
VSR	C		√										√			
NORM	A				√								√			
ARITHMETIC CODES																
DAD	A	√	√		√	√							√	√		
TAD	A	√	√		√	√							√	√		
VAD	A	√	√		√	√							√	√		
DSU	A	√	√		√	√							√	√		
BDSU	A	√	√		√	√							√	√		
VSU	A	√	√		√	√							√	√		
BVSU	A	√	√		√	√							√	√		

OP-CODE	OPERAND - 1						OPERAND - 2									
	Address/ Constant	Push/Down	Index	Indirect	Memory		Address/ Constant	Push/Down	Index	Indirect	Memory					
	A/C	P	*	i	E	F	A/C	P	*	i	E	F	MPAC	OVFIND	ABORT	SEE
ARITHMETIC CODES Cont.																
DMP	A	√	√		√	√							√			
DOT	A	√	√		√	√							√	√		
VXSC	A	√	√		√	√							√			
DDV	A	√	√		√	√							√	√		
BDDV	A	√	√		√	√							√	√		
V/SC	A	√	√		√	√							√	√		
VXV	A	√	√		√	√							√	√		
VPROJ	A	√	√		√	√							√	√		
VXM	A		√		√	√							√	√		
MXV	A		√		√	√							√	√		
MISCELLANEOUS																
PUSH		√														
SETPD	C															
SSP	A		√		√		C									
STADR																

2.2.3 Memory Layout

Since our "machine" is actually a program, it must occupy memory. Locations 6000_8— 7672_8, most of bank ϕ, some of bank 1, and some Fixed/Fixed memory are reserved for the Interpreter itself. Other areas of memory are set aside for use by the Interpreter and no other programs. Five VAC (Vector Accumulator) Areas, which are five Push-down Lists each requiring 43 registers, occupy memory locations 431_8 through 777_8. Registers in locations 100_8 through 137_8 may be used exclusively by interpretive programs for temporary storage. Five sets of 12 special "hardware" registers such as MPAC, ADRLOC, PUSHLOC, and QPRET are located in addresses 140_8 through 264_8 for simultaneous use by a maximum of five interpretive programs.

We represent memory by the three groups called "local erasable," "high memory," and "low memory." Local erasable memory consists of non-switchable erasable locations 61_8 to 1377_8 plus the current E-bank we are in (see footnote, page 2-4). Since it is assumed that all calculations can be accomplished within non-switchable erasable and one E-bank, interpretive programs will not change E-Banks. Fixed interpretive memory is composed of two "half-memories." Low memory is composed of fixed-switchable banks 4_8 through 17_8 and high memory, of fixed-switchable banks 21_8 through 37_8.

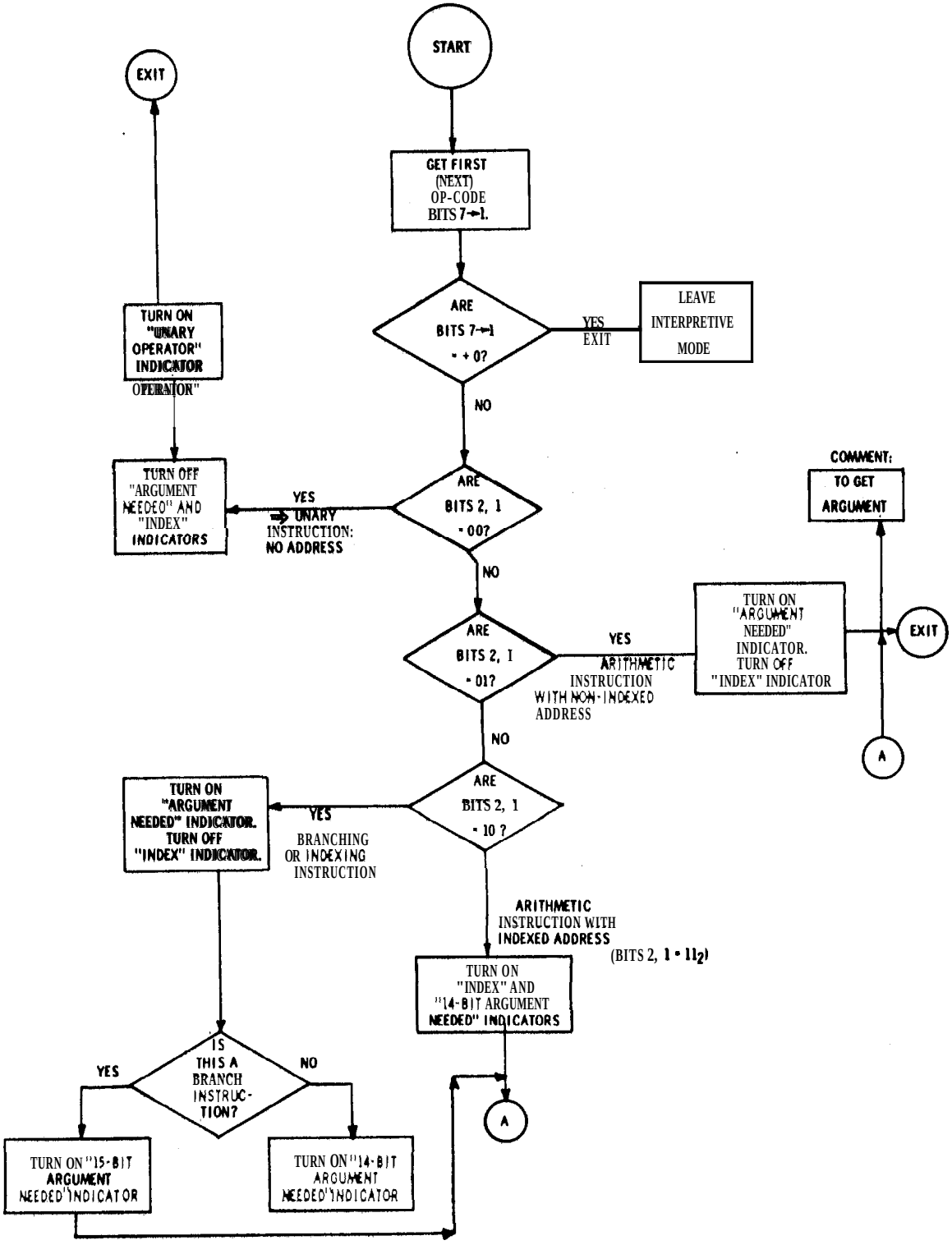


Half-Memories

(The cross-hatched area may not be used by interpretive programs.)

Variables may be stored anywhere in erasable memory locations other than $0—77_8$, and programs may be stored anywhere in high or low memory. Because the address of a branching instruction has 15 bits for definition, interpretive programs may branch to any other program anywhere in memory. Programs stored in low memory, however, may refer to constants stored only in low memory, while programs in high memory must refer to constants stored only in high memory.

INTERPRETIVE OP-CODE SELECTION LOGIC



2.3 Addressing

Although our general instruction format provides a full 15-bit word-length for the definition of operand addresses, we rarely have more than 14 bits available with which to define an argument. As we discussed under Instruction Representation, the class of indexed instructions which takes an argument (bits 2 and $1 = 11_2$) uses bit 15 of the argument to specify which index register will be used. Thus, only 14 bits are left for defining magnitude. Also, the class of indexable but unindexed instructions which takes an address (bits 2 and 1 = 01) uses a zero in bit 15 *of* **a** specified argument to indicate that an address is not required from the Push-down List, Push-up arguments are, therefore, indicated by a 1 in bit 15, found in the op-code word that lies where the address would otherwise have been.

To summarize then, only addresses of branching instructions may use a full 15-bit word for definition. All other arguments must be contained with 14 magnitude bits and thus reference but that part of memory (low or high) in which the current interpretive program resides.

Class 01_2, which consists of arithmetic instructions which take non-indexed arguments, may address any location in local erasable or in the half-memory from which the instruction was taken. Thus, if the E-Bank is set to 6 for this interpretive program and we wish to execute the instruction STORE X at location 17000_8, *X* must be located in general erasable 61_8——1377_8 or in E-Bank 6 (3000_8——3377_8).

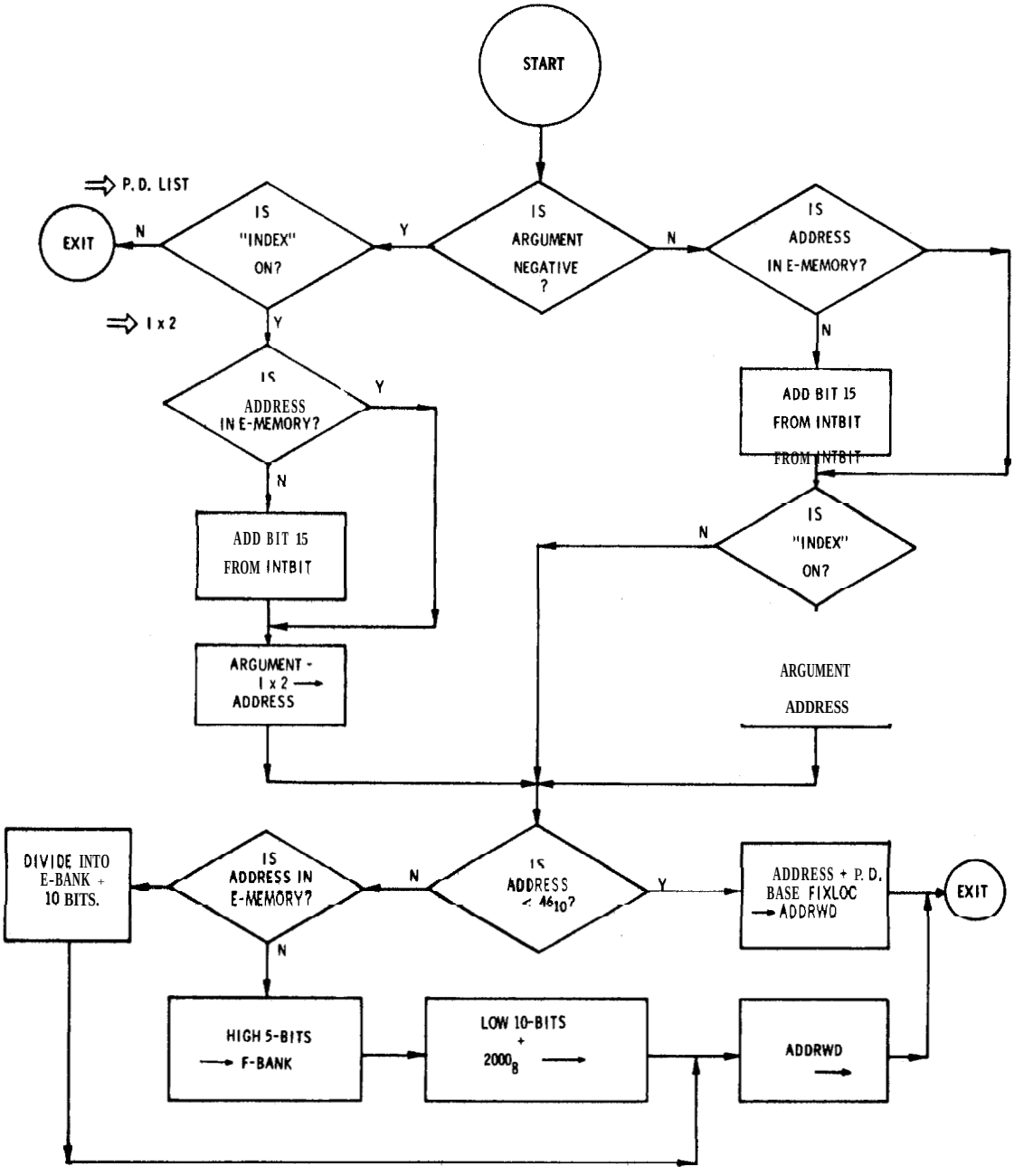
Class 10_2, which contains branching and index-manipulating instructions taking non-indexed operands, may address all of interpretive memory with the branching instructions. The index-manipulatirzg instructions may refer to erasable; some use their addresses as operands.

Class 11_2, which consists of indexed arithmetic instructions requiring arguments, may address any location in local erasable or in the half-memory in which the instruction is located. The arguments of instructions in this class will of course be modified by subtracting the contents of either index register 1 or 2 to yield net operand addresses. For example, if the E-Bank has been set to 4 prior to processing the interpretive program, and if the instruction DAD X, 1 is located at the high half-memory address 57643_8, then the net operand address **X,** minus the contents of index register 1, must be located in erasable locations 61_8——1377_8, in E-Bank 4 (2000_8——2377_8), or in high memory (banks 21_8——37_8).

Since an argument limited to defining an address within 14 bits lacks space to refer to locaticns in high memory (octal locations 51024 —— 107777), when a program enters the Interpretive Mode, the configuration of bit 15 (1 for high and 0 for low) is stored in the INTBIT15 register and will be appended to all 14-bit addresses. Thus if INTBIT15 contains a 0, we are programming in low memory and need no more than 14 bits to define operands. If INTBIT15 contains a 1, we are operating in high memory, Since the Interpreter appends the INTBIT15 configuration of 1 to all high memory addresses, we require only the low-order 14 bits to specify any high memory address.

Because our sequence-changing or branching instructions may refer to both half-memories, we may not modify their arguments through index registers. We use, instead, a system of indirect addressing. If the address of a branching instruction refers to erasable, the contents of this erasable location are construed to be the address of the next interpretive instruction, If this address is also located in erasable, however, we take its contents to be the address of the next instruction. We continue in this manner until we find a fixed-memory address, which is processed as the next instruction. Thus if at erasable location X sits the quantity 1307_8, then the instruction GOTO X results in transferring control to the instruction located at the erasable address 1307_8, as long as the contents of 1307_8 is an address in fixed memory.

ADDRESS SELECTION LOGIC

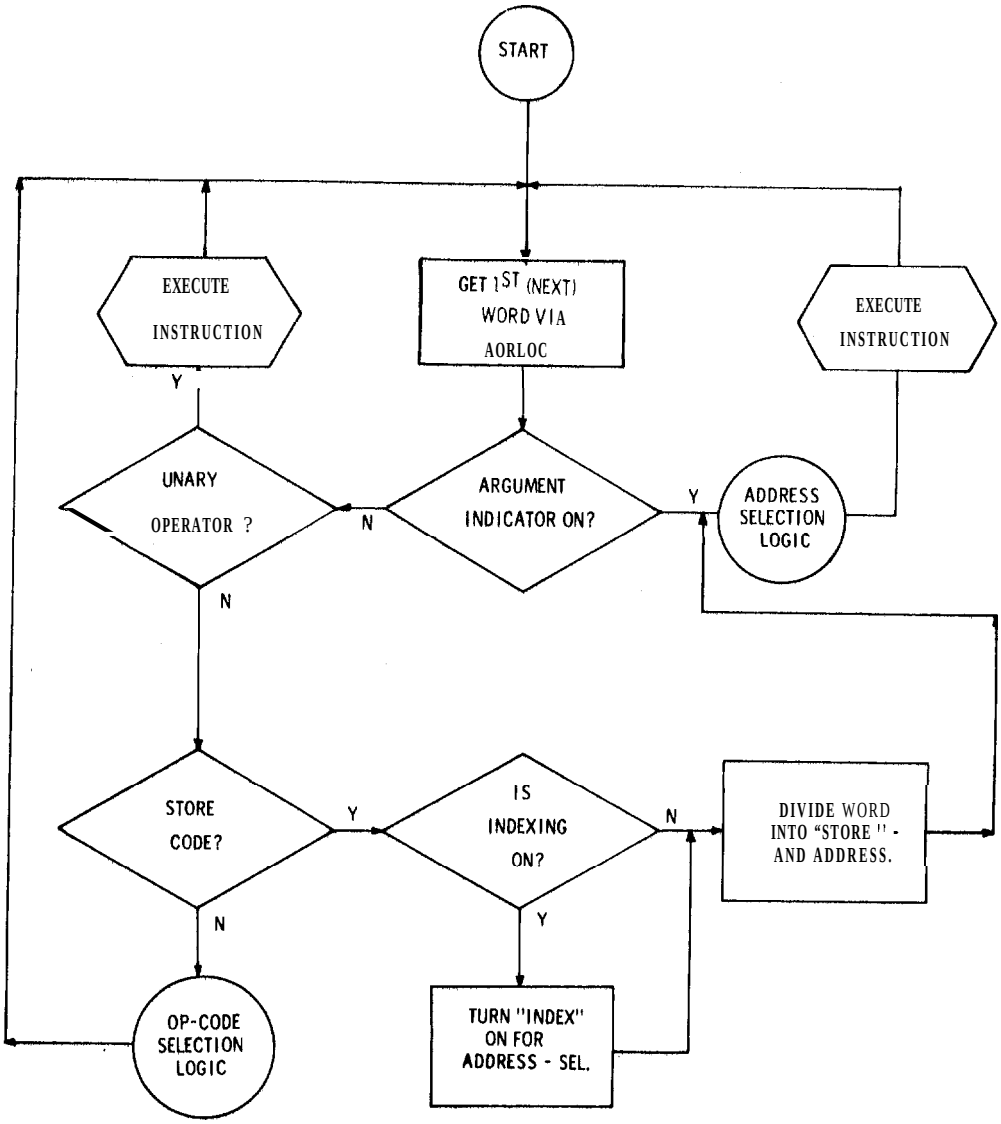


2.4 The Dispatcher (INTERPRETIVE CPU)

The Dispatcher is the software Central Processing Unit of the Interpreter. Originally, a programmer must use the instruction TC INTPRET to gain access to the Dispatcher. If an interpretive program is interrupted, however, for the basic processing of a higher priority job, for instance, then we may later return to the Dispatcher via the EXECUTIVE program with a TC DANZIG call.

The Dispatcher looks at the first word in an interpretive program and decides whether it contains an operation code or an address. If the word contains two operation codes, it divides the word into its two 7-bit components and sends the first to the op-code selection logic. If the op-code requires an address, the Dispatcher looks at the next word, sending it to the address selection logic if it is an address. If it is not an address and the op-code requires an argument, the Dispatcher fetches one from the Push-down List and sends that to the address selection logic, The Dispatcher finally executes the instruction, If the first word had contained a second op-code, it would have been treated in the same manner. When an instruction has been executed, the Dispatcher is ready to process the next word.

Not shown nor previously discussed is the concept of Mode. At all times the Interpreter must know if it is dealing with single-, double-, or triple-precision operators, or vector or matrix operators, These modes are determined by the particular op-code which is being processed. Some op-codes set the mode while others require that the mode be set by previous op-codes. Actually, the programmer need not usually concern himself with the mode, as during his programming the mode will logically behave itself in accordance with his logical needs. There follows a chart which indicates the behavior of the mode according to the sundry interpretive instructions, page 2-29.


START
EXECUTE INSTRUCTION
GET 1ST (NEXT) WORD VIA AORLOC
EXECUTE INSTRUCTION
Y
UNARY OPERATOR ?
N
ARGUMENT INDICATOR ON?
Y
ADDRESS SELECTION LOGIC
N
STORE CODE?
Y
IS INDEXING ON?
N
DIVIDE WORD INTO "STORE" - AND ADDRESS.
N
Y
OP-CODE SELECTION LOGIC
TURN "INDEX" ON FOR ADDRESS - SEL.

2.5 The Push-Down List

As we mentioned in the Introduction, the Push-down List is a means of saving memory by using implied rather than direct addresses to reference temporary storage. It also aids the programmer in allowing the machine itself to temporarily store quantities. The list may contain 38 15-bit quantities with the distinction that the last quantity entered (pushed down) is the first to be withdrawn (pushed up). For example, we would process the equation $x = ab + cd - ef$, as follows:

$$x = ab + cd - ef$$

Operation	Push-Down List after Operation
1) Form the product ab and push it down.	ab — • • •
2) Form the product cd and add to it ab from the Push-down List.	— — • • •
3) Push-down the sum ab + cd and form the product ef.	ab + cd — • • •
4) Subtract the product ef from the Push-down List. Store the difference in x.	x = ab + cd - ef

The Push-down Location Register functions as a pointer for the Push-down List in the same manner as the Z-Register acts as a pointer for AGC instructions. Initialized at 0 by the EXECUTIVE program, the contents of PUSHLOC are increased by 1 as MPAC quantities are stored and pushed down word by word into the fish-down List. Thus, we would process the equation

$$x = \frac{a^2 + b^2}{c^2 + d^2}$$

as follows:

1) Form a^2 and push it down.	a^2	PUSH-
	———	LOC
	...	
2) Form b^2, and add to it a^2 from the Push-down List. Push down the sum.	$a^2 + b^2$	PUSH-
	———	LOC
	...	
3) Form c^2 and push that down.	c^2	PUSH-
	$a^2 + b^2$	LOC
	...	
4) Form d^2 and add to it the last quantity entered in Push-down List,	$c^2 + d^2$	PUSH-
	———	LOC
	...	
5) Divide the sum of $c^2 + d^2$ into Push-down List. Store quotient in **x.**	———	PUSH-
	———	LOC
	...	
	$x = \frac{a^2 + b^2}{c^2 + d^2}$	

When d^2 is taken from the Push-down List in order to be added to c^2, a "push-up" operation is performed which causes the contents of the location at which FUSHLOC is pointing to come out of the Push-down List and into the MPAC. After each push-up operation, the contents of PUSHLOC are decreased by one. The contents of PUSRLOC may be set or changed by a programmer with the instruction SETPD X (set PUSHLOC) which will cause the contents of PUSHLOC to be set equal to X (normally $0 \leq X \leq 42_{10}$) *so* as to point to a slot in the Push-down list.

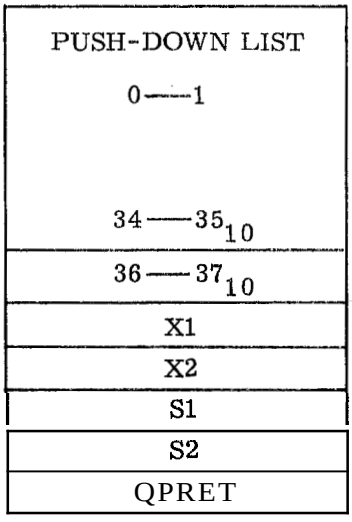
Whenever an op-code requires an argument and one is not specified, the last quantity entered into the Push-down Lost is automatically pushed-up into MPAC to be used as the operand. We may push down quantities from the MPAC, however, only with 3 instructions:

PUSH	The contents of MPAC are stored in the Push-down location whose address is in PUSHLOC.
PDDL	The contents of MPAC are stored in the Push-down List. MPAC is then loaded in DP with the quantity at X.
PDVL	The contents of MPAC are stored in the Push-down List. MPAC is then loaded with the vector at X.

Quantities in the Push-down List never physically move up or down; only the pointer PUSHLOC moves as a result of having its contents increased or decreased.

Each of 5 interpretive jobs has a 43-word work area associated with it. Within the work area, or VAC area, is the Push-down List in locations $0\text{-}37_{10}$, the two index registers X1 and **X2,** the two step registers S1 and **S2,** and the QPRET register. All 43 registers are available as push-down area if the programmer does not need X1 — QPRET.

VAC AREA



The following instructions affect the mode of an operand. Wherever pertinent, the mode is given for: meaningful mode as input to the op-code (mode-in); what mode the op-code operates under (op-mode); and, how the mode is left upon completion of the op-code (mode-out).

Instruction	Mode-In	Op-Mode	Mode-Out
VLOAD		V	V
TAD	DP, TP	TP	TP
SIGN	-	DP	-
VXSC	DP, TP, V	V, DP	V
CGOTO	-	SP	-
TLOAD	-	TP	TP
DLOAD	-	DP	DP
V/SC	DP, TP, V	V, DP	V
SLOAD	-	SP	DP
SSP	-	SP	-
PDDL	-	DP	DP
MXV	V	MATRIX	-
PDVL	-	V	V
CCALL	-	SP	-
VXM	V	MATRIX	-
NORM	DP, TP	SP	-
DMPR	DP, TP	DP	DP
DDV	DP, TP	DP	-
BDDV	DP, TP	DP	-
VAD	V	V	-
VSU	V	V	-
BVSU	V	V	-
DOT	V	V	DP
VXV	V	V	-
VPROJ	V	V	-
DSU	DP, TP	DP	-
BDSU	DP, TP	DP	-

Instruction	Mode-In	Op-Mode	Mode-Out
DAD	DP, TP	DP	-
DMP	DP, TP	DP	-
SETPD		SP	-
VSL1—8, VSR1—8	-	V	-
SL1—4, SR1—4	-	DP, TP	-
SL1—4R, SR1—4R	-	DP	-
SIN, COS, ASIN, ACOS	DP, TP	-	-
SQRT, DSQ, DCOMP, ABS	DP, TP	-	-
ROUND	DP, TP	-	DP
VDEF	DP, TP	-	V
VSQ	V	-	DP
UNIT	V	-	-
VCOMP	V	-	-
ABVAL	V	-	DP

2.6 The Instructions

As we stated under Instruction Representation, we divide the 127 interpretive instructions into 8 categories according to what the instructions generally do. As each group is discussed, it will be helpful to refer to the instruction chart between pages 2-11 and 2-18 for a concise summary of the characteristics of each instruction set.

2.6.1 Memory Load and/or Store Instructions

This group of instructions merely transfers data to and from storage locations.

STORE X transfers the double-precision, triple-precision, or vector contents of the Multi-Purpose Accumulator (MPAC) to the E-memory location specified by X, where X may be an indexed or direct address. A double-precision MPAC quantity would be stored in X and X + 1; a triple-precision quantity in X, X + 1, and X + 2; and a vector quantity in X through X + 5.

S, D, T, or V-LOAD instructions are all concerned with loading the MPAC with some quantity stored in location X. We have the option of loading MPAC with a single-precision quantity (SLOAD X), a DP quantity (DLOAD X), a TP quantity (TLOAD X), or a vector quantity (VLOAD X). If we load MPAC with an SP quantity, we clear the two MPAC registers following the register containing the SP number to allow for later arithmetic computations. Similarly, we load a DP quantity into MPAC such that the register following the two containing the DP number is cleared. Loading MPAC with either an SP or DP quantity sets the store mode to DP. TLOAD and VLOAD simply load MPAC with a TP or vector quantity and set the store mode to TP or vector, respectively. The location X from which a quantity is loaded may be a direct, indexed, or push-up address for all load instructions except SLOAD, which requires that X be either direct or indexed.

It is often convenient to combine the abilities to store and load into one operation. We therefore have an instruction which stores the DP, TP, or vector quantity located in MPAC into memory locations starting at X and reloads MPAC with the quantity at location Y. After storing MPAC, we may load it with either a DP quantity (STODL X) in TP form *so* that the first and second registers in MPAC contain
(Y)

the DP quantity and the third is cleared of its previous contents, or we may reload it with a vector quantity (STOVL X). Reloading MPAC with a DP quantity sets the store mode to DP while reloading it with a vector sets the store mode to vector. The memory address X may be either indexed or direct, and the address *Y* may be indexed, direct, or push-up.

We also combine storing operations with the capacity to call a subroutine: STCALL X. Without changing the store mode, the DP, TP, or vector quantity in MPAC is stored into memory starting at location X, and the subroutine at *Y* is called while the return address of the location after the second address is left in the QPRET register. Both the storage address X and the address Y from which we call a subroutine must be direct addresses.

A specialized use of the store operation is the function of the instruction STQ X which stores the contents of the QPRET register into one 15-bit word at the erasable location X. We would want to save QPRET in this manner if we wished to call a routine within a subroutine. X would have to be an erasable location since we would later reference it with a GOTO X *to* return from a secondary subroutine, via indirect addressing.

2.6.2 Control Instructions

These instructions contain the branching operations which bring about changes in the sequence of instructions. All of the following instructions with the exception of EXIT and RVQ take a direct address. If any direct address except one taken by RTB, which branches to basic language, refers to erasable memory, it is interpreted as an indirect address. See Addressing: Indirect Addressing, page 2-22.

As we mentioned in the Introduction, the return address register QPRET is the interpretive counterpart of the Q-Register in the AGC. It contains the address at which we shall continue processing upon return from a subroutine, Further details of its use will be discussed with the instructions below.

The branching instruction GOTO X initiates a sequence change which will cause instruction processing to be resumed at the address X. The contents of QPRET are unaffected, GOTO is a right-hand operation code, meaning that if it is in the left-hand position of an op-code pair, the right-hand op-code must be blank. A

variation of this instruction is the Computed GOTO, or (CGOTO X,) which is an in-
(Y)
dexed GOTO instruction. The contents of the erasable location X are added to the fixed address *Y*. Instruction execution will resume at whatever location is referenced by the sum Y + S(X). Like GOTO, CGOTO is a right-hand op-code.

The CALL X instruction calls the subroutine beginning at location X and leaves a return address in QPRET. The Computed Call (CCALL X) is the indexed
(Y)
form of CALL, causing a branch in instruction execution to the location referenced by the sum *Y* + S(X). CCALL differs from CGOTO in that CCALL leaves a return address in QPRET.

Two interpretive instructions provide for return from a subroutine initiated by a CALL (or CCALL) instruction. If the subroutine itself contains no CALL or CCALL instructions, a Return Via QPRET (RVQ) will effect a resumption of instruction execution at the address left in QPRET. If, however, a subroutine is to be called with a CALL or CCALL instruction in the midst of processing another subroutine, QPRET must be stored temporarily with an STQ X as discussed above. Upon completion of subroutine processing, a GOTO X will provide a return to the current program.

If, for some reason, a transition from interpretive to basic language is desired, a Return to Basic (RTB X) instruction will cause basic instruction execution to begin at the fixed memory location X. The exit from the subroutine via a TC Q will return control to the Interpreter. *

If, however, a more prolonged departure from the Interpreter is necessary than that implied with an ~~RTB,~~ an Exit from Interpreter instruction (EXIT) is available. If EXIT is in the left-hand position of a pair of op-codes, basic instruction execution begins at the word after the EXIT instruction. If EXIT is in the right-hand position of a pair of op-codes, basic instruction execution begins at the word following the last address used by the left-hand op-code. EXIT is a right-hand operation code.

2.6.3 Decision Instructions

This group consists of the branching instructions which cause sequence changes upon testing the results of arithmetic operations.

* TC DANZIG is always a safe return from basic, if Q is not to be trusted.

Within this category is a subgroup of instructions which effects a GOTO X if the TP quantity in MPAC is greater than, equal to, or less than 0. The Branch Plus instruction (BPL X) causes a GOTO X if the TP number in MPAC is greater than or equal to 0. Branch Zero (BZE X) branches to X if the MPAC TP quantity is equal to 0, and Branch Minus (BMN X) branches to X if the TP quantity is less than 0. Otherwise, in all these cases, no operation occurs.

By testing the single-precision quantity in MPAC for a configuration equal to 0, we may cause a GOTO X with a Branch High Order Zero instruction (BHIZ,). If the SP quantity in MPAC is unequal to 0, no operation occurs.

In the Introduction, we briefly mentioned the existence of an OVFIND register which records overflow caused by a number of instructions. Two interpretive instructions interrogate the state of the register for use by the current program:

BOV X	Branch an Overflow
BOVB X	Branch an Overflow to Basic

No operation occurs if the overflow indicator is off; i.e., the contents of OVFIND are equal to + 0. If the contents of OVFIND are equal to ± 1, however, OVFIND is reset to + 0 and BOV becomes the instruction GOTO X while BOVB becomes the instruction RTB X, where X is a fixed-memory address (for BOVB only). The Executive Program initializes OVFIND to + 0 at the beginning of every new interpretive job.

2.6.4 Switch Instructions

Since many on-off indicators are required by Apollo lunar missions, four erasable locations are set aside to contain 120 switches numbered 0—119D. Fourteen instructions test and manipulate the switches. Every instruction but 3 effects two levels of operation: first, it may set the switch to 1, clear it to 0, invert it (0 becomes 1; 1 becomes 0), or cause no-operation; secondly, it may branch if the switch was initially on, branch if the switch was initially off, branch unconditionally, or cause no operation.

The fourteen instructions easily divide themselves into pairs.

SET X sets switch X (to 1) while SETGO $\frac{X}{Y}$ sets switch X and branches to Y. If Y references erasable memory it is construed as an indirect address.

BONSET X and BOFSET X set switch X and branch to Y if X was initially on
Y Y
or off, respectively.

CLEAR X and CLRGO X set switch X and CLRGO branches unconditionally
Y
to Y.

BONCLR X and BOFCLR X clear switch X and branch to Y if X was initially
Y Y
on or off respectively.

INVERT and INVGO invert switch X and INVGO branches unconditionally to *Y*.
Y
BONINV X and BOFINV X invert switch X and branch to *Y* if X was initially
Y Y
on or off respectively.

BON X and BOF X branch to Y if X is on or off respectively.
Y Y

Two address words are required by all of the above instructions except SET, CLEAR, and INVERT.

2. 6. 5 Index Register Instructions

Two 15-bit index registers (X1 and X2) may be used for simple arithmetic computations with single-precision numbers as well as for address modification. The number of the index register (1 or 2) involved with an index register operation follows any of the 10 different instructions and is separated from the instruction by a comma, AXT, 1 for example refers to X1 while AXT, 2 indicates that X2 is involved. Six different operations load and store the two index registers.

AXT (1, 2) X loads the single-precision constant X into the specified index register X1 or X2. Similarly the instruction AXC (1, 2) X loads the complement of the SP quantity X into the specified index register. Examples of a single-precision constant would be an interpretive address or an octal or decimal constant. Under no condition are the contents of X loaded into X1 or X2 by an AXT or AXC.

LXA (1, 2) X loads the specified index register with the contents of the erasable register X, while LXC (1, 2) X, which loads the specified index register with the complement of the contents of erasable location X can complement the index register with LXC, 1, X1 or LXC, 2, X2.

SXA (1, 2) X simply stores the contents of the specified index register in erasable register X. XCHX (1, 2) X exchanges the contents of the specified index register with the contents of the erasable location X.

Three different instructions modify the contents of index registers.

INCR (1, 2) X adds any single-precision constant X to the contents of the specified index register,

XAD (1, 2) X adds the contents of the erasable location X to the contents of the specified index register, and

XSU (1, 2) X subtracts the contents of erasable location X from the contents of the specified index register,

Besides the two index registers which accompany every interpretive job are two 15-bit step registers (S1 and S2) which may be used as temporary storage but which are principally designed to count. Along with the index registers, they are used to count with the TIX instruction: TIX (1, 2) X (Count and branch on index). If a difference greater than zero may be obtained by reducing the contents of the specified index register (X1 and X2) by the contents of its corresponding step register, then the reduced value replaces the index and the instruction GOTO X is executed. If the difference is equal to or less than zero, no operation occurs. Thus, loop-control may be initiated by pre-setting a step-register to some decrement and using a TIX at a loop-decision point in a program.

2.6.6 MPAC Manipulation Instructions

These instructions manipulate data in the Multi-Purpose Accumulator without affecting memory.

We combine two of the Loading instructions with a push-down operation. PDDL X pushes down the DP, TP, or vector quantity located in MPAC into the push-down list and reloads MPAC with the DP quantity located at X. The register following the two containing the DP number is cleared, and the store mode is set DP. The memory address X may be direct, indexed, or push-up. We may vary the above instruction by reloading MPAC with a vector quantity (PDVL X), thereby setting the store mode to vector.

The instruction SIGN X is a DP sign test, where X must reference erasable memory. If the DP quantity at location X and X + 1 is equal to or greater than zero, no operation occurs, If the DP quantity is less than zero, however, and if the store mode is DP or TP, the TP quantity in MPAC is replaced by its complement. If the DP quantity is less than zero and the store mode is vector, the vector contents of MPAC are replaced by their complement.

The scalar function DP Square Root (SQRT) causes the TP quantity in MPAC to be replaced by the square root of the DP quantity in MPAC; i.e., the initial contents of MPAC are normalized, the DP square root of the normalized number computed, and that result unnormalized in accordance with the original normalizing shift, *so* that MPAC + 2 has marginal significance. Receiving an argument less than -10^{-4} causes an abort, The DP Square instruction (DSQ) causes the TP quantity in MPAC to be replaced by the DP quantity in MPAC, squared. The Square of Vector Length instruction (VSQ) causes the square of the absolute value of the vector quantity in MPAC to become a TP MPAC quantity, thus changing the store mode to DP. If the absolute value of the vector quantity in MPAC is greater than or equal to 1, we set OVFIND and leave an overflow-corrected result in MPAC.

ROUND TO DP (ROUND) causes the TP quantity in MPAC to be rounded to DP *so* that the first two registers in MPAC contain the DP number and the third register is cleared. If overflow occurs, OVFIND is set and the overflow-corrected result + 0 is left in MPAC.

A Triple-Precision Complement instruction (DCOMP) causes the TP quantity in MPAC to replaced by its complement. Similarly, the Vector Complement instruction (VCOMP) replaces the vector quantity in MPAC with its complement.

The Triple-Precision Absolute Value instruction (ABS) causes the triple precision quantity in MPAC to be replaced by its absolute value. The Vector Length instruction (ABVAL) replaces the absolute value of the vector quantity in MPAC with a TP quantity, thereby changing the store mode to DP. Furthermore, the vector quantity in MPAC, squared, replaces the DP contents of push-down location 34D. If the absolute value of the vector in MPAC is less than 2^{-21}, then the result is zero. If the absolute value of the MPAC vector quantity is greater than or equal to 1, OVFIND is set to indicate an unspecified result. The Unit Vector Function instruction (UNIT) causes the vector in MPAC to be replaced by the quotient of the MPAC vector divided by twice the absolute value of the MPAC vector. Also, the absolute value

of the MPAC vector quantity, squared, replaces the contents of 34D in DP form, and the absolute value of the MPAC vector replaces 36D in DP form, OVFIND is set if the absolute value of the vector quantity is less than 2^{-21} or if it is greater than or equal to 1, in which case the result is incorrect.

Vector Define (VDEF) pushes up for V_Y and again for V_Z so that the DP quantity in MPAC, the DP quantity in V_Y and the DP quantity in V_Z becomes the vector contents of MPAC, setting the store mode to vector.

The scalar function DP Sine (SIN[SINE]) replaces the TP quantity in MPAC with the product of 0.5 and the sine of (2π multiplied by the DP quantity in MPAC). The scalar function DP Cosine (COS[COSINE]) replaces the TP quantity in MPAC with 0.5 multiplied by the cosine of (the product of 2π and the DP contents of MPAC).

The DP Arc-sine instruction (ARCSIN[ASIN]) replaces the TP quantity in MPAC with 2π multiplied by the Arc-sine of twice the DP contents of MPAC. This is the inverse of the SIN function. Receipt of an argument greater than 0.5001 in magnitude causes an abort. The DP Arc-cosine function (ARCCOS[ACOS]) replaces the TP contents of MPAC with $\frac{1}{2\pi}$ multiplied by the Arc-Cosine of twice the DP contents of MPAC. This is the inverse of COS. As with ASIN, receipt of an argument whose magnitude exceeds 0.5001 induces an abort.

The TP contents of MPAC may be shifted right or left 1 through 4 times by a SCALAR SHIFT instruction. SCALAR SHIFT RIGHT (SR1 through SR4) replaces the TP quantity in MPAC with the product of the MPAC TP quantity and 2^{-j} where j may be 1 through 4. Shifting a quantity right one place is obviously the equivalent of dividing the quantity by 2. SCALAR SHIFT LEFT (SL1 through 4) replaces the TP contents of MPAC with the product of the TP quantity in MPAC and 2^{+j} where j equals 1, 2, 3, or 4. Of course, shifting a quantity left one place is the equivalent of multiplying the quantity by 2. If significant bits are lost, we set OVFIND but leave the overflow-corrected result as the TP contents of MPAC. We have the option of rounding with the above instructions, thus creating the instructions Scalar Shift Right and Round (SR1R, SR2R, SR3R, SR4R) in which the TP quantity in MPAC multiplied by 2^{-j}, where j equals 1-4, is rounded to a DP number, X, and is followed by a word of +0, which replaces the TP contents of MPAC. Likewise, we have a Scalar Shift Left and Round instruction (SL1R - SL4R) which rounds the TP MPAC quantity multiplied by 2^{+j} to a DP number, X, and replaces the TP contents of MPAC with X followed by a +0 word.

The General Vector Shift instruction (VSR X) replaces each component of the vector quantity in MPAC by its original value multiplied by a 2^{-X} and rounded to DP form, If X is an indexed address and the resulting address is negative, execute a VSL - X instead. X must be greater than zero and less than 29 if it is a direct address and if indexed, it must be greater than -128 and less than 128. Similarly, the General Vector Shift Left instruction (VSL X) replaces each component of the vector in MPAC by its original component multiplied by 2^{X}. Upon overflow of any component, OVFIND is set and the overflow-corrected result is left in MPAC. If the address is indexed and the resulting address is negative, execute a V J - X instead. X must be greater than 0 and less than 28 if it is a direct address.

General Scalar Shift Right and Left instructions exist which take direct or indexed addresses. General Scalar Shift Right (SR X) replaces the TP quantity in MPAC with the MPAC TP quantity multiplied by 2^{-X} where X is greater than -42 and less than 42. X can be negative only if the address was indexed. X must be greater than 0 and less than 42 if it is a direct address, and if indexed, X_S must be greater than -128 and less than 128. X_S is the stored address before index modification; in all cases, X is the net address. When overflow occurs, OVFIND is set and the overflow-corrected result is left in MPAC. General Scalar Shift Left (SL X) has the same format as SR X except that the TP quantity in MPAC multiplied by 2^{X} replaces the MPAC TP quantity. Again we have the option to include the capacity to round in the above instructions. General Scalar Shift Right and Round (SRR X) is the same as SR except that the TP quantity in MPAC multiplied by 2^{-X} is rounded to a DP number which replaces the TP contents of MPAC, with a word equal to +0 following the DP quantity. X must be greater than 0 and less than 29 if it is a direct address. General Scalar Shift Left and Round (SLR X) is the same as SL except that the TP quantity in MPAC multiplied by 2^{X} is rounded to a DP number which replaces the TP contents of MPAC with a word equal to +0 occupying the third register in MPAC. X must be greater than 0 and less than 14 if it is a direct address.

Vector Shift Right (1—8) may also include the capacity to round quantities. Vector Shift Right and Round (VSR1—VSR8) replaces each component of the MPAC vector by its original value multiplied by 2^{-j} (where j = 1—8) and rounded to a DP quantity. Vector Shift Left (VSL1—VSL8) replaces each component of the vector in MPAC by its original contents multiplied by 2^{+j} where j = 1—8. If overflow occurs in any component, OVFIND is set and the overflow-corrected result is left in MPAC.

In situations where we know we have enough bits to define the result of an arithmetic computation, we still may not know whether we will have any — and how many — leading zeroes. Instead of shifting the result left once and testing OVFIND ourselves, we use the instruction SCALAR NORMALIZE (NORM) to give us maximum precision by shifting the MPAC TP quantity left N number of times. Provided the TP quantity is not zero, the triple-precision contents of MPAC are shifted 2^N until greater than or equal to 0.5. The complement of the number of shifts-left (-N) is stored in the specified operand location X. The TP MPAC quantity multiplied by 2^N replaces the TP contents of MPAC. If the TP quantity in MPAC is 0, however, -0 goes into the specified operand location X and the triple precision contents of MPAC are left unchanged,

2.7 Arithmetic Instructions

This group of instructions uses operations involving both memory locations and MPAC. As before, the programmer must keep track of the imaginary point in his computations since he is manipulating his quantities in the fixed point accumulator format with the binary point falling between bits 15 and 14.

An ADD instruction enables us to add the quantity at memory location X to the contents of MPAC, replacing the previous contents of MPAC with the new sum. We have the option to add in DP, TP, or vector form. DP Add (DAD X) replaces the DP contents on MPAC with the sum of the MPAC quantity and the DP quantity at X, which may be a direct, indexed, or push-up address. If overflow results, OVFIND is set, and the overflow-corrected result is left in MPAC. TP Add (TAD X) replaces the triple-precision contents of MPAC with the sum of the MPAC quantity and the TP quantity at location X. Similarly, Vector A (VAD X) replaces the vector in MPAC with the sum of the MPAC vector and the vector starting at location X. As with DAD, both TAD and VAD set OVFIND upon overflow and leave the overflow-corrected result in MPAC.

The subgroup of Subtract instructions reduces the contents of MPAC by the quantity at location X. In all arithmetic forms, OVFIND is set on overflow and the overflow-corrected result is left in MPAC. DP Subtract (DSU X) reduces the DP contents of MPAC by the DP quantity in memory location X, where X may be direct, indexed, or pushed-up. BDSU X, the DP Subtract From or DP Backwards Subtract instruction is a very convenient instruction which reduces the DP quantity stored at memory location X by the DP contents of MPAC and stores the difference in MPAC. Thus, if we wished to replace the contents of MPAC, which are 2, by the contents of X2 = (10_{10}) minus the quantity in MPAC, we may simply subtract 2 from 10 and store 8 in MPAC. Otherwise, we would have to push-down MPAC and bring the contents of X2 into MPAC before we could calculate a difference of 8.

Vector Subtract (VSU X) reduces the vector in MPAC by the vector at X, leaving the vector difference in MPAC. BVSU X, or Backwards Vector Subtract, serves the same convenience as BDSU in allowing us to subtract the vector contents of MPAC from the vector at X, storing the difference in MPAC. As usual, overflow with either VSU or BVSU causes OVFIND to be set and the overflow-corrected result to be stored in MPAC.

The group of Multiply instructions replaces the contents of MPAC with the quantity at X multiplied by the quantity in MPAC. DP Multiply (DMP X) stores the

product of the DP contents of MPAC and the DP quantity at X in triple-precision form, We have a rounding option with this instruction, giving us DMPR X, DP Multiply and Round which rounds to DP the product of the DP contents of MPAC and the DP quantity at X. The rounded DP product is followed by a word equal to + 0 replaces the TP contents of MPAC.

The Vector Dot Product instruction (DOT X) stores in TP form the product of the vector in MPAC and the vector at X, thus setting the store mode to DP. Upon the overflow of any component, OVFIND is set and the overflow-corrected result is stored in MPAC.

Under VXSC X, or Vector Times Scalar, if the initial store mode is Vector, each component of the vector in MPAC is multiplied by the DP quantity at X, with the rounded products replacing their respective components of the MPAC vector. If the initial store mode is DP or TP, it is changed to Vector, and each component of the vector at X is multiplied by the DP quantity in MPAC to form the vector in MPAC, as above.

DDV X (DP Divide By) and BDDV X (Backwards DP Divide) are our two divide instructions. If the absolute value of the DP contents of MPAC is less than the DP quantity at X; i. e., if the divisor is larger than'the dividend, then DDV X divides the DP contents of MPAC by the DP quantity at X to yield a DP quotient which will be stored along with a word equal to + 0 in MPAC. If overflow results, OVFIND is set and ± 0.99999999 is left in the DP contents of MPAC. The Backwards Divide is the same as DDV X, except that the DP quantity at X will be the dividend, and the DP contents of MPAC will be the divisor, as long as the DP quantity in MPAC is larger than the DP quantity at X.

The Vector Divided by Scalar instruction (V/SC X) divides each component of the vector in MPAC by the DP quantity at X if the store mode is set to Vector, Each of the DP quotients replaces its respective vector components of MPAC. If the initial store mode is DP or TP, it is changed to Vector, and each component of the vector at X is divided by the DP quantity in MPAC to form an MPAC vector. If overflow occurs in any component, the operation is terminated with OVFIND set and unspecified results left in MPAC.

The Vector Cross Product (VXV X) replaces the vector in MPAC with the product of the MPAC vector and the vector at X. If overflow results, OVFIND is set, leaving an overflow-corrected result in MPAC.

VPROJ X (Vector Projection) causes the vector contents of MPAC to be replaced by the product of ⌊V(MPAC) • V(X)⌋ and V(X). OVFIND is set if overflow results, and the result obtained with overflow-corrected ⌊V(MPAC) • V(X)⌋ is left in MPAC.

The Matrix Pre-Multiplication by Vector instruction (VXM X) replaces the vector in MPAC with the product of ⌊V(MPAC) • M(X)⌋. OVFIND is set on overflow, leaving an overflow-corrected result in MPAC.

The last of the arithmetic codes is MXV X, the Matrix Post-Multiplication by Vector instruction. This causes the product ⌊M(X) • V(MPAC)⌋ to replace the vector contents of MPAC. As usual, OVFIND is set on overflow, leaving an overflow-corrected result in MPAC.

2.8 Miscellaneous Instructions

Under this group are classed the four instructions PUSH, SETPD, SSP, and STADR.

PUSH causes the DP, TP, or Vector quantity in MPAC to be pushed down into the Push-down List.

The instruction SETPD(Set PUSHLOC) is discussed on page 2-27. SSP X/Y (Set Single-Precision) replaces the single-precision contents of X with quantity Y. Y may be any arithmetic, logical or address constant.

STADR, as we discussed in the Introduction, distinguishes a positive store code from a positive operand address by causing it to be assembled in complemented form. At execution time, it is recognized as a "STADR'D" code, recomplemented and executed as a STORE X.

DISTRIBUTION LIST

Internal

M. Adams (MIT/GAEC)
J. Alexshun
R. Battin
P. Bowditch/F. Siraco
R. Boyd
R. Byers
G, Cherry (10)
N. Cluett
E. Copps (30)
R. Crisp
J. Dahlen
J. DeLisle
E. Duggan
J. B. Feldman
P. Felleman (5)
S. Felix
J. Flanders (MIT/KSC)
J. Fleming (5)
J. Gilmore
F. Grant
Eldon Hall
T. Hemker (MIT/NAA)
D. Hoag
F. Houston
L. B. Johnson
M. Johnston
A. Kosmala (10)
A. Laats
L. Larson
S. Laquidara (MIT/FOD)
J. Lawrence (MIT/GAEC)
J. Lawson
T. M. Lawton (MIT/MSC)
D. Lickly
F. Martin (10)
G. Mayo
James Miller (10)
John Miller
J. Nevins
J. Nugent
M. Petersen
R. Ragan
J. Rhode
D. Russell
R. Scholten
N. Sears (10)
J. Shillingford
W. Shotwell (MIT/AC)
W. Stameris
J. Suomala
W. Tanner
M. Trageser
R. Weatherbee
R. White
W. Widnall (5)
R. Woodbury
W. Wrigley
Apollo Library (2)
MIT/IL Library (6)

External:

W. Rhine (NASA/MSC)	(2)
NASA/RASPO	(1)
AC Electronics	(3)
Kollsman	(2)
Raytheon	(2)
Major H. Wheeler (AFSC/MIT)	(1)

MSC: (18 + 1R)

National Aeronautics and Space Administration
Manned Spacecraft Center
Apollo Document Distribution Office (PA2)
Houston, Texas 77058

LRC: (2)

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virgin ia
Attn: Mr. A. T. Mattson

GAEC: (3 + 1R)

Grumman Aircraft Engineering Corporation
Data Operations and Services, Plant 25
Bethpage, Long Island, New York
Attn: Mr. E. Stern

NAA: (18 + 1R)

North American Aviation, Inc.
Space and Information Systems Division
12214 Lakewood Boulevard
Downey, California
Attn: Apollo Data Requirements
Dept. 096-340, Bldg. 3, CA 99

NAA RASPO: (1)

NASA Resident Apollo Spacecraft Program Office
North American Aviation, Inc.
Space and Information Systems Division
Downey, California 90241

ACSP RASPO: (1)

National Aeronautics and Space Administration
Resident Apollo Spacecraft Program Officer
Dept. 32-31
AC Electronics Division of General Motors
Milwaukee 1, Wisconsin
Attn: Mr. W. Swingle

Defense Contract Administration (1)
Service Office, R
Raytheon Company
Hartwell Road
Bedford, Massachusetts 01730

Mr. S. Schwartz (1)
DOD, DCASD, Garden City
605 Stewart Avenue
Garden City, L. I., New York
Attn: Quality Assurance

Mr. D. F. Khols (1)
AFPRO (CMRKKA)
AC Electronics Division of General Motors
Milwaukee 1, Wisconsin 53201