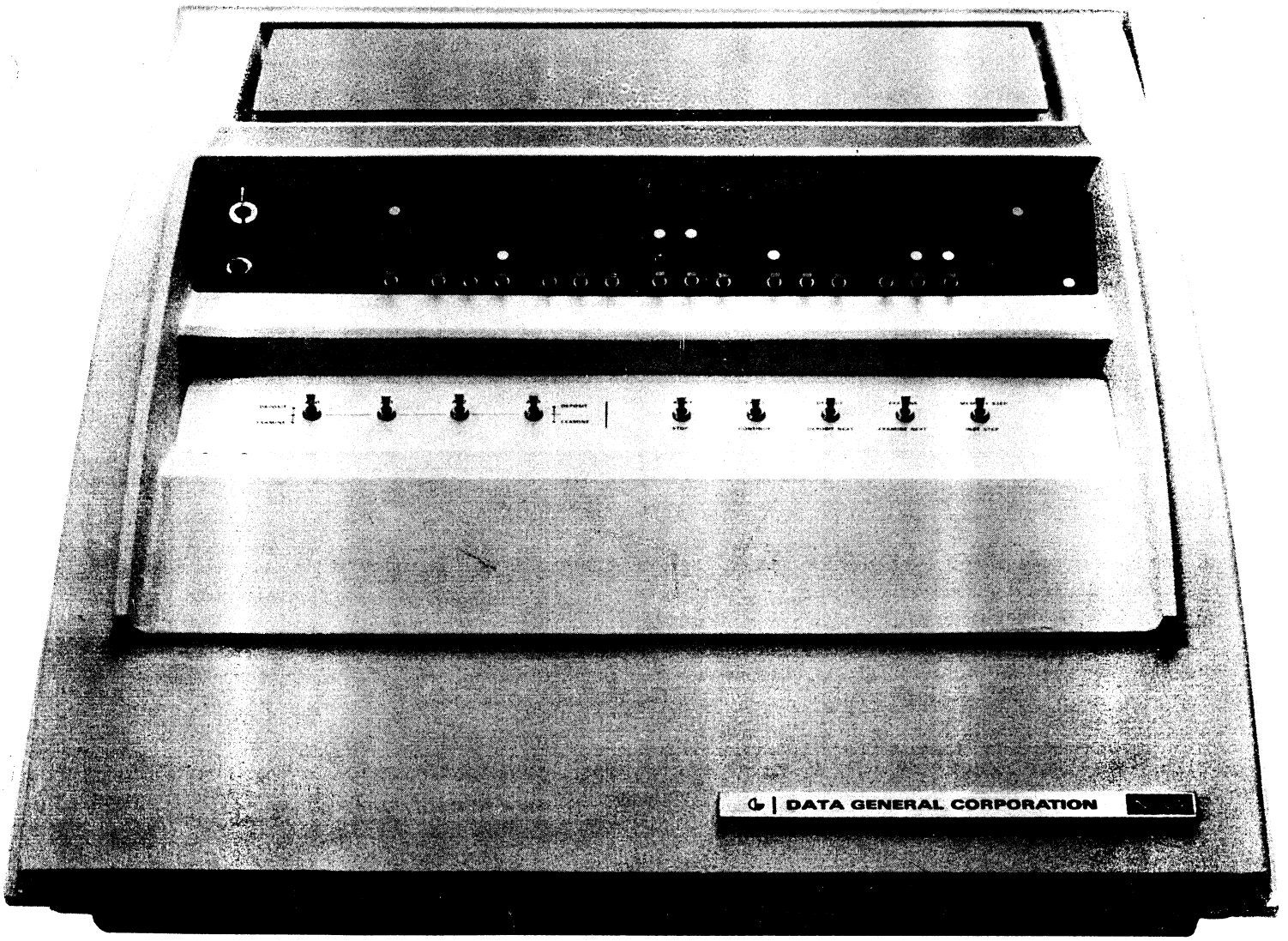
How to use the NOVA.



How to use the NOVA

The NOVA System Reference Manual

DIRECT COMMENTS CONCERNING THIS MANUAL TO

DATA GENERAL CORPORATION • SOUTHBORO, MASSACHUSETTS

November 1968

Written for Data General Corporation by William English.

Printed in the United States of America

NOTE

This manual is the beginning of what will become a complete system reference manual for the NOVA. The present chapters explain the programming of the central processor and the teletypewriter. Additional chapters treating the peripheral equipment and other manuals describing software and maintenance will be issued during the coming months. If you would like to receive such material, simply fill out and mail this form to

Data General Corporation
Southboro, Mass. 01749

Name _____ Organization _____

Address _____

_____ Zip _____

Position _____

Contents

1 INTRODUCTION 1-1

1.1 Instructions 1-3
Instruction format 1-4

1.2 Memory 1-6

2 CENTRAL PROCESSOR 2-1

2.1 Memory Reference Instructions 2-2
Move data instructions 2-5
Modify memory instructions 2-6
Jump instructions 2-7

2.2 Arithmetic and Logical Instructions 2-10
Carry, shift and skip functions 2-12
Arithmetic and logical functions 2-14
Programming examples 2-17

2.3 Input-Output 2-21
Special code-77 functions 2-26

2.4 Program Interrupt 2-30

2.5 Data Channel 2-37

2.6 Processor Options 2-38
Real time clock 2-38
Power monitor and autorestart 2-39

2.7 Operation 2-40

3 HARDCOPY EQUIPMENT 3-1

3.1 Teletypewriter 3-1

APPENDICES

C Floating Point Arithmetic C1

D Instruction Mnemonics D1
Numeric listing D2
Alphabetic listing D5

E In-out Codes E1
In-out devices E2
Teletype code E4

Chapter I
Introduction

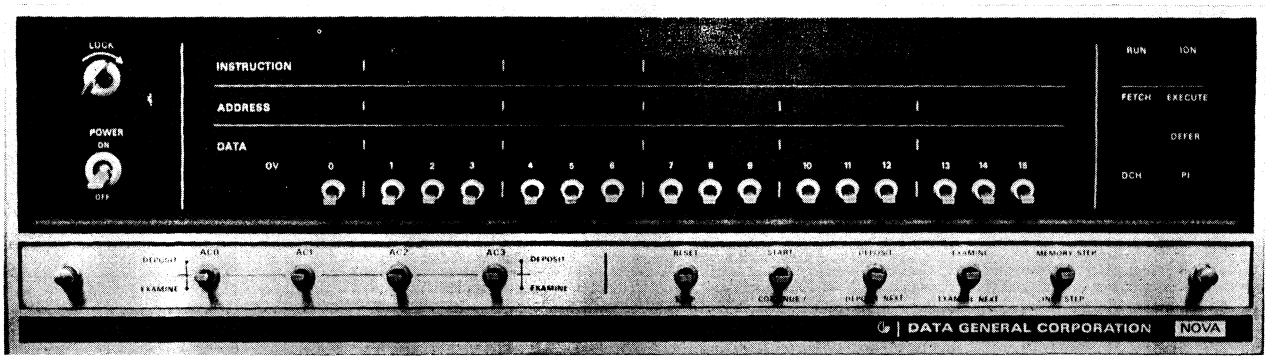
The NOVA is a general purpose computer system that utilizes a 16-bit word length. The machine is organized around four accumulators, two of which can be used as index registers. This accumulator/index register organization provides greater efficiency and ease in programming.

The computer may have both alterable core memory and read-only memory. With the console removed, the system can be operated as a hard-wired controller, whose functions can be altered simply by substituting different read-only memories.

Shown opposite is the table model with a memory capacity of up to 16K words and a teletypewriter for input-output. The processor-memory requires only 5¼ inches when mounted in a standard 19-inch rack. Processor options include real time clock and power failure detector. Available peripheral equipment includes teletypewriter, high speed paper tape reader and punch, card reader, line printer, incremental plotter, display, magnetic tape, magnetic disk, A-D and D-A conversion equipment, and data communications equipment.

The central processor is the control unit for the entire system: it governs all peripheral in-out equipment, performs all arithmetic, logical, and data handling operations, and sequences the program. It is connected to the memory by a memory bus and to the peripheral equipment by an in-out bus. The processor handles words of sixteen bits, which are stored in a memory with a maximum capacity of 32,768 words. The bits of a word are numbered 0 to 15, left to right, as are the bits in the registers that handle the words. Registers that hold addresses are fifteen bits, numbered according to the position of the address in a word, *ie* 1 to 15. Words are used either as computer instructions in a program, as addresses, or as operands, *ie* data for the program. The program can interpret an operand as a logical word, an address, a pair of 8-bit bytes, or a 16-digit signed or unsigned binary number. The arithmetic instructions operate on fixed point binary numbers, either unsigned or the equivalent signed numbers using twos complement conventions.

The processor performs a program by executing instructions retrieved from consecutive memory locations as counted by the 15-bit program counter PC. At the end of each instruction PC is incremented by one so that the next instruction is normally taken from the next consecutive location. Sequential program flow is altered by changing the contents of PC, either by incrementing it an extra time in a test skip instruction or by replacing its contents with the value specified by a jump instruction. The other internal registers of importance to the programmer are four 16-bit accumulators, AC0 to AC3. Data can be moved in either direction between any memory location and any accumulator. Although a word in memory can be incremented or decremented, all other arithmetic and logical operations are performed on operands in the accumulators, with the result appearing in an accumulator. Associated with the accumulators is the Carry flag, which indicates when a carry occurs out of bit 0 in an arithmetic instruction. The left and right halves of any accumulator can be swapped, the contents of any accumulator can be tested for a skip, and the 17-bit word contained in any accumulator combined with Carry can be rotated right or left. An instruction that references memory can address AC2 or AC3 as an index register, and transfers to and from peripheral devices are also made through the accumulators.



Operator Console

On the processor console is a set of data switches through which the operator can supply words and addresses to the program. The console also has a number of control switches that allow the operator to start and stop the program, to deposit the contents of the data switches in any memory location or accumulator, and to display the contents of any location or accumulator in the data lights (these indicators display the contents of the memory buffer through which all transfers are made between memory and accumulators). The address lights display the contents of PC, the instruction lights display the left half of the instruction word currently being executed. The remaining lights display the Carry flag and a number of internal control conditions that are useful in program debugging.

Any instruction that references memory may address AC2 or AC3 as an index register. Instructions that move data to and from memory or the peripherals address a single accumulator as a source or destination of data while addressing a memory location or an in-out device. But the arithmetic and logical instructions do not have to reference memory; they simply address two accumulators, either or both of which may supply operands, and one of which may receive the result. Thus memory is used for storage of the program and permanent data, but all calculations are carried out in the accumulators and intermediate results are held right in them. This reduces considerably the amount of data movement as compared with a single accumulator system, and thus saves instructions. For example, in as trivial an operation as exchanging the contents of two memory locations A and B, the multi-accumulator organization reduces the time by one third.

Exchange with one accumulator	*Exchange with two accumulators*
A→AC	A→AC1
AC→TEMP	B→AC2
B→AC	AC1→B
AC→A	AC2→A
TEMP→AC	
AC→B	

Since an arithmetic or logical instruction does not contain a memory address, there are many bits that can be used for functions other than specifying the basic operation and the operands: the same instruction that adds or subtracts can also shift the result or swap its halves, test the result and/or carry for a skip, and specify whether or not the result shall actually be retained. Hence the percentage of time saved increases with the complexity of the program.

And there are advantages other than speed. The system is much more convenient to use, programming is much easier because the data being processed is much handier. The accumulators and their associated logic are essentially like the pad one uses at one's desk, whereas the memory fulfills the function of a set of reference books and a notebook kept on one's side. The results of address calculations are immediately available for index purposes to the memory reference instructions. One accumulator can be used for in-out data transmission without disturbing others being used continually for computations. Complex software routines such as multiplication, division and floating point can be performed without constantly referencing memory.

The input-output hardware allows the program to address up to sixty-two devices. A single instruction can transfer a word between an accumulator and a device and at the same time control the device operation. Included in the in-out system are facilities for program interrupts and high speed data transfers. The interrupt system facilitates processor control of the peripheral equipment by allowing any device to interrupt the normal program flow on a priority basis. The processor acknowledges an interrupt request by storing PC in location 0 and executing the instruction addressed by the contents of location 1. A high speed device, such as magnetic tape or disk, can gain direct access to memory through a data channel without requiring the execution of any instructions; the program simply pauses while access is made. The data channel logic allows the transfer of data to or from memory, incrementing of a memory word, and adding external data to a word already in memory. The latter two features allow such functions as signal averaging and pulse height analysis.

1.1 INSTRUCTIONS

The types of functions performed by instructions in most computers are the following.

1. Move data between memory and the operating registers.
2. Modify memory, usually in conjunction with a test to determine whether to alter the program sequence.
3. Alter the program sequence by jumping to a new location.
4. Perform an arithmetic or logical operation.
5. Test the value of a word or flag, or one word against another, to determine whether to alter the program sequence.
6. Transfer data to or from the peripheral equipment.

In many computers the first and fourth and the third and fifth groups overlap. In the NOVA groups 1 and 3 are unique. But groups 4 and 5 coincide: every arithmetic and logical instruction can test the result for a skip.

The following lists the registers that must be specified and the functions performed by the various instruction classes in the NOVA.

Move data	One memory location, one accumulator. Either may be the source of the operand, the other is the destination.
Modify memory	One memory location. Increment or decrement contents; skip if result is zero.
Jump	One memory location from which the next instruction is taken. A return address can be saved in AC3.
Arithmetic and logic	Two accumulators. One or both may be source of operand(s). Perform arithmetic or logical function, with a bit-0 carry affecting the Carry flag as indicated. If desired, swap halves of answer or rotate it with Carry one place right or left, load result into either accumulator, and skip on condition specified for result and/or Carry.
Input-output	One accumulator, one IO device. Transfer word in either direction between any accumulator and one of up to three registers in up to sixty-two devices. Also operate device as specified.

Note: A subclass of these instructions executes no transfer and specifies only a device. The instruction either operates the device or skips on a selected condition in it.

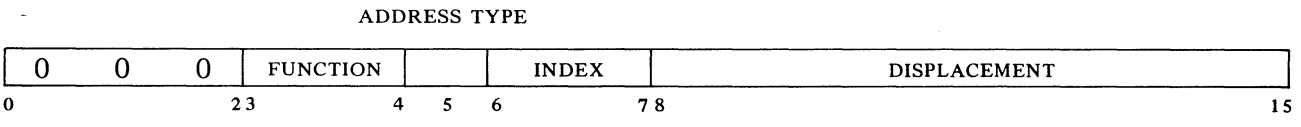
Addressing. Instructions in the first three classes must address a memory location. Each instruction word contains information for determining the effective address, which is the actual address used to fetch or store the operand or alter program flow. The instruction specifies an 8-bit displacement which can directly address any location in four groups of 256 locations each. The displacement can be an absolute address, *ie* it may be used simply to address a location in page zero, the first 256 locations in memory. But it can also be taken as a signed number that is used to compute an absolute address by adding it to a 15-bit base address supplied by an index register. The instruction can select AC2 or AC3 as the index register; either of these accumulators can thus be used as an ordinary index register to vary the address computed from a constant displacement, or as a base register for a set of different displacements. The program can also select PC as the index register, so any instruction can address 256 words in its own vicinity (relative addressing).

Now the computed absolute (15-bit) address can be the effective address. However, the instruction can use it as an indirect address, *ie* it can specify a location to be used to retrieve another address. Bits 1–15 of the word read from an indirectly addressed location can be the effective address or they can be another indirect address.

Automatic Incrementing and Decrementing. The program can make use of an automatic indexing feature by indirectly addressing any memory location from 00020 to 00037 (addresses are always octal numbers). Whenever one of these locations is specified by an indirect address, the processor retrieves its contents, increments or decrements the word retrieved, writes the altered word back into memory, and uses the altered word as the new address, direct or indirect. If the word is taken from locations 00020–00027, it is incremented by one; if taken from locations 00030–00037, it is decremented by one.

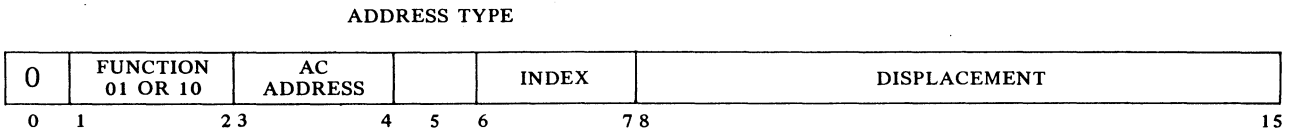
Instruction Format

There are four basic formats for instruction words. In all but the arithmetic and logical instructions, bit 0 is 0. If bits 1 and 2 are also 0, bits 3 and 4 specify the function (jump or modify memory) and the rest of the word supplies information for calculating the effective address. Bits 8–15 are the displacement, bits 6 and 7 specify the index register if any, and bit 5 indicates the type of addressing, direct or indirect.



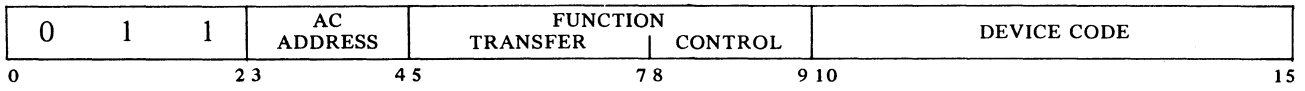
JUMP AND MODIFY MEMORY FORMAT

If bits 1 and 2 differ they specify a move data function. Bits 3 and 4 address an accumulator, the rest of the word is as above.



MOVE DATA FORMAT

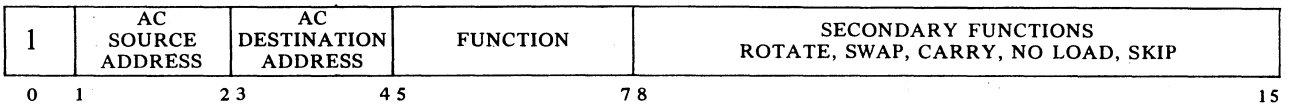
Bits 1 and 2 both being 1 indicate an in-out instruction. In this case the function is specified by bits 5–9, of which bits 5–7 indicate the direction of transfer and select one of three registers in the device. The transfer takes place between the accumulator addressed by bits 3 and 4 and the device selected by bits 10–15. Bits 8



IN-OUT FORMAT

and 9 of the function part specify an action to be performed, such as starting the device. If bits 5–7 are all 0 or all 1, there is no transfer and bits 8 and 9 specify a control or skip function respectively.

If bit 0 is 1, bits 5–7 specify an arithmetic or logical function. One operand is taken from the accumulator addressed by bits 1 and 2; a second operand, if any, from that addressed by bits 3 and 4. The rest of the word specifies the other functions that can be performed, including whether or not the result is to be loaded into the destination accumulator.



ARITHMETIC AND LOGIC FORMAT

The NOVA assembly program recognizes a number of mnemonics and other initial symbols that facilitate constructing complete instruction words and organizing them into a program [*Appendix D*]. In particular there are three-letter mnemonics for the 2- and 3-bit functions; these mnemonics also represent whatever bits are constant for the class the instruction is in. *Eg* the modify memory mnemonic

ISZ

assembles as 010000, the arithmetic mnemonic

SUB

assembles as 102400.

NOTE

Throughout this manual all numbers representing instruction words, register contents, codes and addresses are always octal, and any numbers appearing in program examples are octal unless other-

wise specified. Computer words are represented by six octal digits wherein the left one is always 0 or 1 representing the value of bit 0. The ordinary use of numbers in the text to specify quantities of objects, such as words or locations, to count steps in an operation, or to specify word or byte lengths, bit positions, etc. employs standard decimal notation.

Characters are suffixed to the basic mnemonic to specify the control part of an IO function and most of the secondary functions in the arithmetic and logical class. The displacement and addresses of accumulators and index registers are separated from the mnemonic by a space and from each other by commas. Anything written at the right of a semicolon in a program listing is commentary that explains the program but is not part of it.

1.2 MEMORY

From the addressing point of view, the entire memory is a set of contiguous locations whose addresses range from zero to a maximum dependent upon the capacity of the particular installation. In a system with the greatest possible capacity, the largest address is octal 77777, decimal 32,767. But the memory is actually made up of a number of core memory modules, each having a capacity of 1024, 2048 or 4096 words, and can also contain read-only memory modules. The latter may be used for storage of pure (unalterable) programs and constants; they usually contain 1024 words but may be of any size. An address supplied by the program is actually decoded in two parts, the more significant part to select a memory module and the less significant to select a location within that module, but this need not concern the programmer. From the point of view of the programmer, memory modules of different sizes and types differ only in speed and the fact that the contents of the read-only memory cannot be altered electrically. Common arithmetic and in-out routines are available in standard read-only memory modules; others are available on a custom basis.

The basic processor cycle time depends on the type of memory as follows (actual memory access times are less).

	Time in microseconds
1K Memory	6.5
2K Memory	3.9
4K Memory	2.6
Read-only Memory	2.4

Memory Allocation. The use of certain locations is defined by the hardware.

0–1	Program interrupt locations
20–27	Autoincrementing locations
30–37	Autodecrementing locations

Chapter II
Central Processor

This chapter describes all NOVA instructions but does not discuss the special effects of the in-out instructions when they address specific peripheral devices. The chapter treats the memory reference instructions and the arithmetic and logical instructions in detail, presents a general discussion of input-output, and describes the effects of the in-out instructions on processor elements, including the program interrupt and the real time clock. Effects of in-out instructions on particular peripheral devices are discussed with the devices in the remaining chapters.

In the description of each instruction, the mnemonic, name, and time are at the top, the format is in a box below them. The mnemonic assembles to the word in the box, where bits in those parts of the word represented by letters assemble as 0s. The letters indicate portions that must be added to the mnemonic to produce a complete instruction word.

The processor execution time given at the top assumes reference to a 4K memory with cycle time 2.6 μs for both the instruction and the operand, if any. At the end of each description the time is given as a function of the processor cycle time *M* as listed in § 1.2.

Twos Complement Conventions. The signed numbers used as displacements in referencing memory and as operands for the arithmetic instructions utilize the twos complement representation for negatives. In a word or byte used as a signed number, the leftmost bit represents the sign, 0 for positive, 1 for negative. In a positive number the remaining bits are the magnitude in ordinary binary notation. The negative of a number is obtained by taking its twos complement, with the sign bit included in the operation as though it were a more significant magnitude bit. If x is an n-digit binary number, its twos complement is 2^n-x, and its ones complement is $(2^n-1)-x$, or equivalently $(2^n-x)-1$. Subtracting a number from 2^n-1 (*ie*, from all 1s) is equivalent to performing the logical complement, *ie* changing all 0s to 1s and all 1s to 0s. Therefore, to form the twos complement one takes the logical complement — usually referred to merely as the complement — of the entire word including the sign, and adds 1 to the result. A displacement of 89 and its negative would look like this in bits 8–15 of an instruction word where bit 8 is the sign.

$+89_{10} = +131_8 =$ | 01 011 001 |
8 15

$-89_{10} = -131_8 =$ | 10 100 111 |
8 15

The same numbers used as operands in the accumulators would look like this.

$+89_{10} = +131_8 =$ | 0 000 000 001 011 001 |
0 15

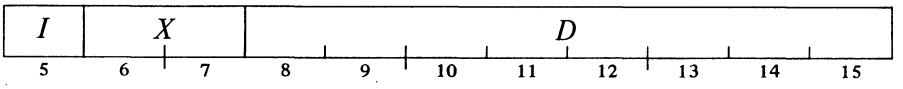
$-89_{10} = -131_8 =$ | 1 111 111 110 100 111 |
0 15

Bit 0 is now the sign and bits 1–8 are not significant. It is thus evident that expanding an integer into a full word is accomplished simply by filling out the word to the left with the sign.

Zero is represented by a number containing all 0s; complementing this number produces all 1s, and adding 1 to that produces all 0s again. So there is only one zero representation and its sign is positive. Moreover there is one more negative number than there are nonzero positive numbers. Hence there are 256 displacements in an octal range −200 to +177. (The most negative number has a 1 in only the sign position.)

2.1 MEMORY REFERENCE INSTRUCTIONS

Bits 5–15 have the same format in every memory reference instruction whether the effective address is used for storage or retrieval of an operand or to alter program flow. Bit 5 is the indirect bit, bits 6 and 7 are the



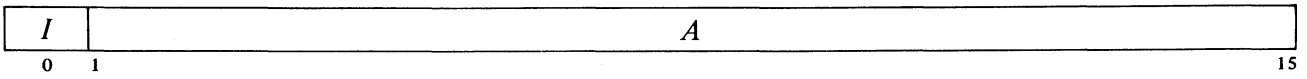
index bits, and bits 8–15 are the displacement. The effective address *E* of the instruction depends on the values of *I*, *X*, and *D*. If *X* is 00, *D* addresses one of the first 256 memory locations, *ie D* is a memory address in the range 00000–00377. This group of locations is referred to as page zero.

If *X* is nonzero, *D* is a displacement that is used to produce a memory address by adding it to the contents of the register specified by *X*. The displacement is a signed binary integer in twos complement notation. Bit 8 is the sign (0 positive, 1 negative), and the integer is in the octal range −200 to +177 (decimal −128 to +127). If *X* is 01, the instruction addresses a location relative to its own position, *ie D* is added to the address in PC, which is the address of the instruction being executed. This is referred to as relative addressing. If *X* is 10 or 11 respectively, it selects AC2 or AC3 as a base register to which *D* is added.

X	*Derivation of address*
00	Page zero addressing. *D* is an address in the range 00000–00377.
01	Relative addressing. *D* is a signed displacement (−200 to +177) that is added to the address in PC.
10	Base register addressing. *D* is a signed displacement (−200 to +177) that is added to the address in AC2.
11	Base register addressing. *D* is a signed displacement (−200 to +177) that is added to the address in AC3.

If *I* is 0, addressing is direct, and the address already determined from *X* and *D* is the effective address used in the execution of the instruction. Thus a memory reference instruction can directly address 1024 locations: 256 in page zero, and three sets of 256 in the octal range 200 less than to 177 greater than the address in PC, AC2 and AC3. If *I* is 1, addressing is indirect, and the processor retrieves another address from the location

specified by the address already determined. In this new word bit 0 is the indirect bit: bits 1–5 are the effective address if bit 0 is 0; otherwise they specify a location for yet another level of address retrieval. This



process continues until some referenced location is found with a 0 in bit 0; bits 1–15 of this location are the effective address *E*.

If at any level in the effective address calculation an address word is fetched from locations 00020–00037, it is automatically incremented or decremented by one, and the new value is both written back in memory and used either as the effective address or for the next step in the calculation depending on whether bit 0 is 0 or 1. Addresses taken from locations 00020–00027 are incremented, those from locations 00030–00037 are decremented.

The set of all addresses is cyclic with respect to the operations performed in an effective address calculation; regardless of the true sum or difference in any step, only the low order fifteen bits are used as an address. Hence the next address beyond 77777 is 00000, the next below 00000 is 77777.

CAUTION

Incrementing 77777 or decrementing 00000 changes the state of the indirect bit in the address word stored back in memory.

Specific examples illustrating the various addressing methods are given following the discussion of programming conventions that begins here.

Programming Conventions. All memory reference functions are represented by three-letter mnemonics; *eg*

```
ISZ
```

assembles as 010000. For addressing page zero the displacement is simply an address. Thus

```
ISZ         344
```

assembles as 010344. When this word is executed as an instruction it increments the word in location 00344 and skips the next instruction if the incremented word is zero. For relative or base register addressing the displacement is a twos complement integer.

```
ISZ         -34,2
```

assembles as 011344 (0 001 001 011 100 100), in which bits 8–15 have the same configuration as in the previous example, but this time the instruction specifies a location whose address is 34_8 less than the address in AC2.

The initial symbol @ preceding the displacement places a 1 in bit 5 to produce indirect addressing. The examples given above use direct addressing, but

```
ISZ         @-34,2
```

assembles as 013344 (0 001 011 011 100 100), and produces indirect addressing.

For memory reference with an accumulator, the AC address precedes the memory address information and is terminated by a comma. *Eg*

```
LDA         3,-34,2
```

assembles as 035344 (0 011 101 011 100 100).

The assembler also allows the following addressing conventions. A period represents the current address, *ie* the address of the location containing the instruction being executed. Thus

```
LDA     3,.+6
```

is equivalent to

```
LDA     3,6,1
```

A colon following a symbol indicates that it is a symbolic location name.

```
A:      ADD     2,3
```

indicates that the location that contains ADD 2,3 may be addressed symbolically as A. The assembler assigns a 15-bit value to the label A. When A is used in a statement such as

```
LDA     2,A+6
```

the treatment depends on the value of the expression in which A appears. In this case if A+6<00400 its low order eight bits are simply placed in the displacement part of the instruction word and *X* is set to 00. If A+6 is within range of PC, the indicated location is represented as a displacement relative to PC. Otherwise the assembler indicates an error as location A+6 cannot be directly addressed by the instruction.

Addressing Examples. Suppose the following registers contain the numbers listed.

Register	*Contents*
6	100015
12	000035
15	000017
17	000023
23	000011
AC3	000015

Now if the program executes the instruction

```
LDA     1,6
```

which loads AC1 from location 6, AC1 receives the number 100015. AC1 holds the same number after

```
LDA     1,-7,3
```

is executed (effective address = C(AC3) − 7 = 15 − 7 = 6). But

```
LDA     1,@6
```

which indirectly addresses location 6, which in turn indirectly addresses location 15, which directly addresses location 17, loads 23 into the accumulator. AC1 also contains 23 following execution of

```
LDA     1,@15
```

On the other hand, AC1 contains 17 after

```
LDA     1,15
```

or

```
LDA     1,0,3
```

is executed. Now

```
LDA     1,6,3
```

does not address location 6; it addresses 23 (C(AC3)+6=15+6=23) and thus loads 11 into AC1. Note that addressing an autoincrementing location directly does not alter its contents; AC1 simply receives its contents as an operand. AC1 also receives 11 from

```
LDA     1,23
```

or

```
LDA     1,@17
```

But giving

```
LDA     1,@23
```

or

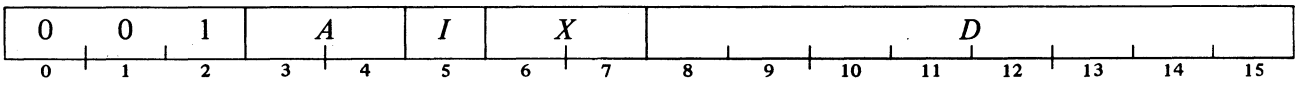
```
LDA     1,@6,3
```

replaces the contents of location 23 with the number 12 and loads 35 (the contents of location 12) into AC1.

Move Data Instructions

These two instructions move data between memory and the accumulators. In the descriptions of all memory reference instructions, *E* represents the effective address. The time given in the top line is for direct addressing, in page zero or relative to PC. Base register addressing requires an additional .3 μs; indirect addressing requires one extra processor cycle time per level; autoincrementing and autodecrementing require no extra time. In the time function given at the end of each instruction, *b* is 1 for base register addressing, otherwise 0.

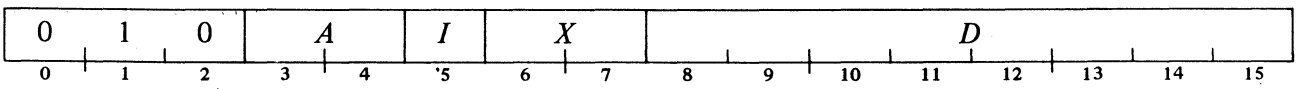
LDA **Load Accumulator** 5.2 μs



Load the contents of location *E* into accumulator *A*. The contents of *E* are unaffected, the original contents of *A* are lost.

Time. $M_{inst} + M_{op} + .3b + \Sigma M_{ind}$ μs

STA **Store Accumulator** 5.5 μs



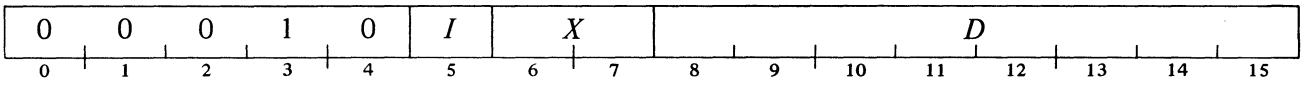
Store the contents of accumulator A in location E. The contents of A are unaffected, the original contents of E are lost.

Time. $M_{inst} + .3 + M_{op} + .3b + \Sigma M_{ind}$ μs

Modify Memory Instructions

These two instructions alter a memory location and test the result for a skip. They are used to count loop iterations or successively modify a word for a series of operations.

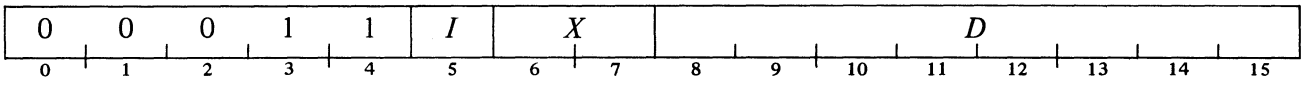
ISZ **Increment and Skip if Zero** 5.2 μs



Add 1 to the contents of location E and place the result back in E. Skip the next instruction in sequence if the result is zero.

Time. $M_{inst} + M_{op} + .3b + \Sigma M_{ind}$ μs

DSZ **Decrement and Skip if Zero** 5.2 μs



Subtract 1 from the contents of location E and place the result back in E. Skip the next instruction in sequence if the result is zero.

Time. $M_{inst} + M_{op} + .3b + \Sigma M_{ind}$ μs

Consider a block of thirty words in locations 2000–2035 that we wish to move to locations 5150–5205 but in reverse order. We could autoincrement through one set, autodecrement through the other, and decrement a control count to determine when the block transfer is complete.

```
        LDA     0,CNT       ;Set up autoincrement location
        STA     0,21
        LDA     0,CNT+1     ;Set up autodecrement location
        STA     0,35
LOOP:   LDA     0,@21       ;Get a word
        STA     0,@35       ;Store it
        DSZ     CNT+2       ;Count down word count
        JMP     LOOP        ;Jump back for next word
        .                   ;Skip to here when count is zero
        .
        .
```

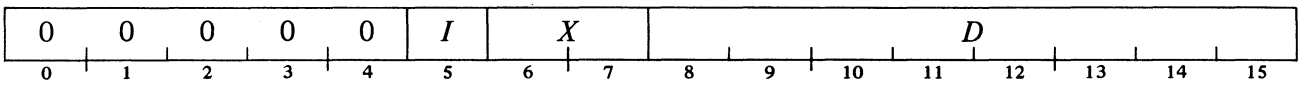
```
CNT:    001777      ;1 before source block
        005206      ;1 after destination block
        000036      ;Word count: 30(10) = 36(8)
```

Of course we could just as well put 177742 (−36) in CNT+2 and replace the DSZ with an ISZ.

Jump Instructions

These two instructions allow the programmer to alter the normal program sequence by jumping to an arbitrary location. They are especially useful for calling and returning from subroutines.

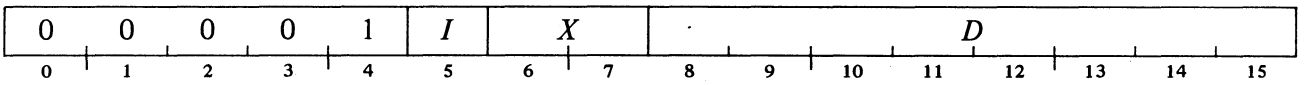
JMP **Jump** 2.6 μs



Load *E* into PC. Take the next instruction from location *E* and continue sequential operation from there.

Time. $M_{inst} + .3b + \Sigma M_{ind}$ μs

JSR **Jump to Subroutine** 3.5 μs



Load an address one greater than that in PC into AC3 (hence AC3 receives the address of the location following the JSR instruction). Load *E* into PC. Take the next instruction from location *E* and continue sequential operation from there. The original contents of AC3 are lost.

NOTE: The effective address calculation is completed before PC+1 is loaded into AC3. Thus a JSR that specifies AC3 as a base register does execute properly; *ie* the previous contents of AC3 are used in the address calculation.

Time. $M_{inst} + .9 + .3b + \Sigma M_{ind}$ μs

The usual procedure for calling a subroutine is to give a JSR whose effective address is the starting location of the routine. Since PC+1 is saved in AC3, a subsequent return can be made to the location following the JSR simply by giving a JMP 0,3. Note also that PC+1 is saved in an accumulator. Hence the subroutine can be reentrant (pure), *ie* memory is not modified by the act of calling it. If we wish to use AC3 in the subroutine, we can store the return address in a convenient place in page zero, say location B, with an STA 3,B and then return with a JMP @B.

A convenient way to handle a number of subroutines that are called frequently is to store their starting addresses in page zero. Suppose we have subroutines starting at locations U, V, W, X, ... If we store these 15-bit addresses at locations UC, VC, WC, XC, . . . respectively in page zero, then we can call a given routine, say the one beginning at X, simply by giving a JSR @XC.

Consider a print subroutine that we wish to use to output fifty words beginning at TAB. The routine begins at PRT, which address is stored in PRTC in page zero. Our main program would contain this.

```
        JSR     @PRTC
        ...                     ;Return here
```

We use AC2 as a base register for counting through the table and AC0 to output the data. The starting address of the table is in TAB1, which is in the vicinity of PRT. The subroutine might look something like this.

```
PRT:    LDA     2,TAB1          ;Set up AC2 as base for table
        LDA     0,0,2           ;Load word for output into AC0
        .                       ;IO part of routine here
        .
        .
        ISZ     PRT+1           ;Increment displacement in load instruction
        DSZ     CNT             ;Done yet?
        JMP     PRT+1           ;No, get next word
        JMP     0,3             ;Yes, return by AC3

TAB1:   TAB
CNT:    62                      ;62(8)=50(10)
```

This routine is incomplete as it destroys itself; to be used again the displacement in location PRT+1 must be changed back to zero. The routine would be faster if we replaced the ISZ with an arithmetic instruction that increments AC2, thus using AC2 as an index register and leaving the LDA displacement alone (it would also be complete as AC2 is set up each time the subroutine is called). It would be even faster if we deleted the ISZ, stored the address TAB–1 in an autoincrementing location, say 23, and loaded AC0 with

```
        LDA     0,@23
```

Argument Passing. Suppose we have an arithmetic subroutine that operates on arguments in AC0 and AC1, leaving the result in AC1. The subroutine call looks like this:

```
        JSR     VS1             ;Call with arguments in AC0, AC1
        ...                     ;Return here with result in AC1
```

and the subroutine looks like this:

```
VS1:    .                       ;Arithmetic operations
        .
        .
        JMP     0,3             ;Return to call + 1
```

In the above the program would have to load the accumulators before calling the routine. Now it is often convenient for the program simply to supply the arguments (or the addresses of the locations that contain them) along with the call and have the subroutine take care of the data transfers. With this version the program gives the arguments in the two memory locations immediately following the JSR,

```
        JSR     VS2
        ...                     ;Argument 1
        ...                     ;Argument 2
        ...                     ;Return here with result in AC1
```

and the return is made to the location following the second argument with the result in AC1.

```
VS2:    LDA     0,0,3       ;Pick up argument 1
        LDA     1,1,3       ;Pick up argument 2
        .
        .
        .
        JMP     2,3         ;Return to call + 3
```

This version is called with the addresses of the arguments following the JSR; otherwise it is the same as version 2.

```
        JSR     VS3
        . . .               ;Address of argument 1
        . . .               ;Address of argument 2
        . . .
        .
        .
        .
VS3:    LDA     0,@0,3      ;Pick up argument 1
        LDA     1,@1,3      ;Pick up argument 2
        .
        .
        .
        JMP     2,3         ;Return to call + 3
```

The next version is the same as version 3 except that the result replaces the second argument in memory.

```
        JSR     VS4
        . . .               ;Address of argument 1
        . . .               ;Address of argument 2 and result
        . . .
        .
        .
        .
VS4:    LDA     0,@0,3      ;Pick up arguments
        LDA     1,@1,3
        .
        .
        .
        STA     1,@1,3      ;Store result
        JMP     2,3
```

The final version is the same as the fourth but AC0 and AC1 are not disturbed by its execution. The call is exactly the same as for VS4.

```
VS5:    STA     0,TM0       ;Save ACs
        STA     1,TM1
        LDA     0,@0,3      ;Pick up arguments
        LDA     1,@1,3
        .
        .
        .
```

Consider a print subroutine that we wish to use to output fifty words beginning at TAB. The routine begins at PRT, which address is stored in PRTC in page zero. Our main program would contain this.

```
        JSR     @PRTC
        ...                     ;Return here
```

We use AC2 as a base register for counting through the table and AC0 to output the data. The starting address of the table is in TAB1, which is in the vicinity of PRT. The subroutine might look something like this.

```
PRT:    LDA     2,TAB1          ;Set up AC2 as base for table
        LDA     0,0,2           ;Load word for output into AC0
        .                       ;IO part of routine here
        .
        .
        ISZ     PRT+1           ;Increment displacement in load instruction
        DSZ     CNT             ;Done yet?
        JMP     PRT+1           ;No, get next word
        JMP     0,3             ;Yes, return by AC3

TAB1:   TAB
CNT:    62                      ;62(8) = 50(10)
```

This routine is incomplete as it destroys itself; to be used again the displacement in location PRT+1 must be changed back to zero. The routine would be faster if we replaced the ISZ with an arithmetic instruction that increments AC2, thus using AC2 as an index register and leaving the LDA displacement alone (it would also be complete as AC2 is set up each time the subroutine is called). It would be even faster if we deleted the ISZ, stored the address TAB–1 in an autoincrementing location, say 23, and loaded AC0 with

```
        LDA     0,@23
```

Argument Passing. Suppose we have an arithmetic subroutine that operates on arguments in AC0 and AC1, leaving the result in AC1. The subroutine call looks like this:

```
        JSR     VS1             ;Call with arguments in AC0, AC1
        ...                     ;Return here with result in AC1
```

and the subroutine looks like this:

```
VS1:    .                       ;Arithmetic operations
        .
        .
        JMP     0,3             ;Return to call + 1
```

In the above the program would have to load the accumulators before calling the routine. Now it is often convenient for the program simply to supply the arguments (or the addresses of the locations that contain them) along with the call and have the subroutine take care of the data transfers. With this version the program gives the arguments in the two memory locations immediately following the JSR,

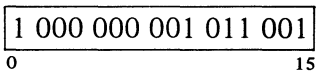
```
        JSR     VS2
        ...                     ;Argument 1
        ...                     ;Argument 2
        ...                     ;Return here with result in AC1
```

```
        STA     1,@1,3      ;Store result
        LDA     0,TM0       ;Restore ACs
        LDA     1,TM1
        JMP     2,3

TM0:    0                   ;Temporary storage for ACs
TM1:    0
```

2.2 ARITHMETIC AND LOGICAL INSTRUCTIONS

To perform logical operations the hardware interprets operands as logical words. For arithmetic operations, operands are treated as 16-bit unsigned numbers, with a range of 0 to $2^{16}-1$. The program however can interpret them as signed numbers in twos complement notation as described at the beginning of this chapter. It is a property of twos complement arithmetic that operations on signed numbers using twos complement conventions are identical to operations on unsigned numbers; in other words the hardware simply treats the sign as a more significant magnitude bit. Suppose an accumulator contains this binary configuration:



As an unsigned number this would be equivalent to

$$100131_8 \quad = \quad 32857_{10}$$

whereas interpreted as a signed number using twos complement notation it would be

$$-77647_8 \quad = \quad -32679_{10}$$

Insofar as processor operations are concerned, it makes no difference which way the programmer interprets the contents of the accumulators provided only that he is consistent.

Numbers in twos complement notation are symmetrical in magnitude about a single zero representation so all even numbers both positive and negative end in 0, all odd numbers in 1 (a number all 1s represents -1). If ones complements were used for negatives, one could read a negative number by attaching significance to the 0s instead of the 1s. In twos complement notation each negative number is one greater than the complement of the positive number of the same magnitude, so one can read a negative number by attaching significance to the rightmost 1 and attaching significance to the 0s at the left of it (the negative number of largest magnitude has a 1 in only the sign position). Assuming the binary point to be stationary, 1s may be discarded at the left in a negative integer, just as leading 0s may be dropped in a positive integer; equivalently an integer can be extended to the left by prefixing 1s or 0s respectively (*ie* by prefixing the sign). In a negative (proper) fraction, 0s may be discarded at the right; as long as only 0s are discarded, the number remains in twos complement form because it still has a 1 that possesses significance; but if a portion including the rightmost 1 is discarded, the remaining part of the fraction is now a ones complement. Truncation of a negative number thus increases its absolute value.

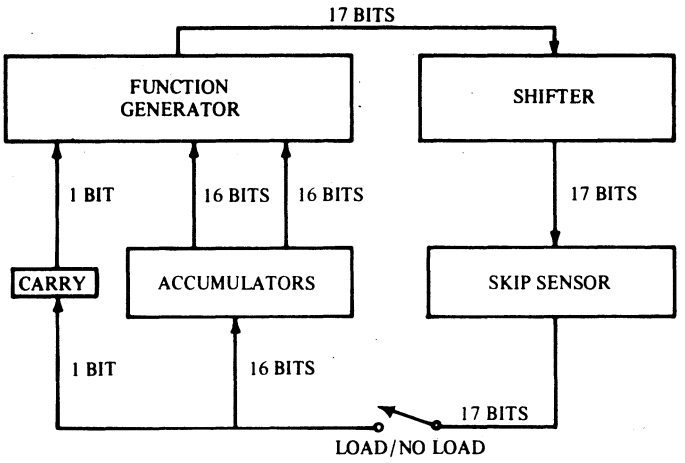
The computer does not keep track of a binary point; the programmer must adopt a point convention and shift the magnitude of the result to conform to the convention used. Two common conventions are to regard a number as an integer (binary point at the right) or as a proper fraction (binary point at the left);

in these two cases the range of signed numbers represented by a single word is -2^{15} to $2^{15}-1$ or -1 to $1-2^{-15}$.

Since each bit position represents a binary order of magnitude, shifting a number is equivalent to multiplication by a power of 2, provided of course that the binary point is assumed stationary. Shifting one place to the left multiplies the number by 2. A 0 should be entered at the right, and no information is lost if the sign bit remains the same — a change in the sign indicates that a bit of significance has been shifted out. Shifting one place to the right divides by 2. Truncation occurs at the right, and a bit equal to the sign must be entered at the left.

Associated with the accumulators is the Carry flag, which is used to detect a carry out of bit 0 in an arithmetic operation. The circumstances that generate a carry out of the most significant bit are obvious when dealing with unsigned numbers. If addition or incrementing increases a number beyond $2^{16}-1$, a carry is produced. In subtraction the condition is the same if instead of subtracting we add the complement of the subtrahend and add 1 to the result (subtraction is performed by adding the twos complement). In terms of the original operands the subtraction $A-B$ produces a carry if $A \geqslant B$. Forming the twos complement of zero generates a carry, for complementing zero produces a word containing all 1s, and adding 1 to that produces all 0s again plus a carry. The statement of the carry conditions in terms of signed numbers is more complex, but they are always exactly equivalent to the conditions given above if the numbers are simply interpreted as unsigned. In any event the complete conditions that produce a carry for numbers signed or unsigned are given in the instruction descriptions.

Arithmetic and Logical Processing. The logical organization of the arithmetic unit is illustrated below. Each instruction specifies one or two accumulators to supply operands to the function generator, which performs the function specified by the instruction. The function generator also produces a carry bit whose value depends upon three quantities: a base value specified by the instruction, the function performed, and the result obtained. The base value may be derived from the Carry flag, or the instruction may specify an independent value.

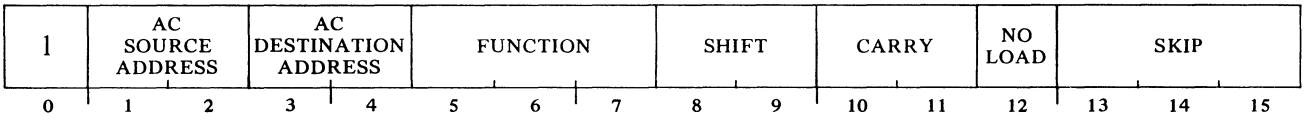


ORGANIZATION OF ARITHMETIC UNIT

The 17-bit output of the function generator, comprising the carry bit and the 16-bit function result, then goes to the shifter. Here the 17-bit result can be rotated one place right or left, or the two 8-bit halves of the 16-bit function result can be swapped without affecting the carry bit. The 17-bit shifter output can then be tested for a skip; the skip sensor can test whether the carry bit or the rest of the 17-bit word is or is not equal to zero. Finally the 17-bit shifted word can be loaded into Carry and one of the accumulators selected by the instruction. Note however that loading is not necessary: an instruction can perform a complicated arithmetic and shifting operation and test the result for a skip without affecting Carry or any accumulator.

Carry, Shift and Skip Functions

An instruction that has a 1 in bit 0 performs one of eight arithmetic and logical functions as specified by bits 5–7 of the instruction word. The function, which may be anything from a simple move to a subtraction, always uses the contents of the accumulator specified by bits 1 and 2; and if a second operand is required, it comes from the accumulator addressed by bits 3 and 4.

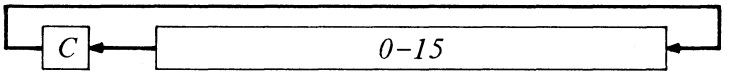
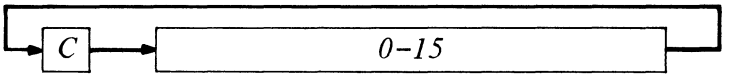


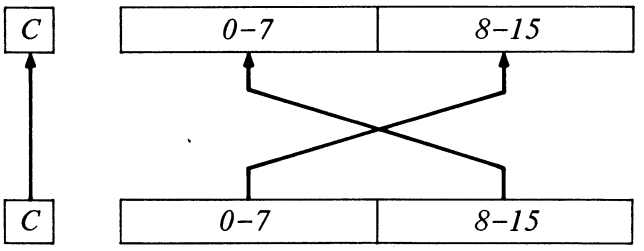
The instruction also supplies a carry bit to the shifter with the result. Bits 10 and 11 specify a base value to be used in determining the carry bit. The instruction supplies either this value or its complement depending upon both the function being performed and the result it generates. The mnemonics and bit configurations and the base values they select are as follows.

Mnemonic	*Bits 10–11*	*Base value for carry bit*
	00	Current state of Carry
Z	01	Zero
O	10	One
C	11	Complement of current state of Carry

The three logical functions simply supply the listed values as the carry bit to the shifter. The five arithmetic functions supply the complement of the base value if the operation produces a carry out of bit 0; otherwise they supply the value given. The carry bit can be used in conjunction with the sign of the result to detect overflow in operations on signed numbers. But its primary use is as a carry out of the most significant bit in operations on unsigned numbers, such as the lower order parts in multiple precision arithmetic.

The 17-bit word consisting of the carry bit and the 16-bit result is operated on by the shifter as specified by bits 8 and 9.

Mnemonic	*Bits 8–9*	*Shift operation*
	00	None
L	01	Left rotate one place. Bit 0 is rotated into the carry position, the carry bit into bit 15.
		
R	10	Right rotate one place. Bit 15 is rotated into the carry position, the carry bit into bit 0.
		
S	11	Swap the halves of the 16-bit result. The carry bit is not affected.



The 17-bit output of the shifter is loaded into Carry and the accumulator addressed by instruction bits 3 and 4 provided bit 12 is 0. A 1 programmed in bit 12 inhibits the loading and prevents the instruction from affecting Carry or the accumulator. Note that it is the shifted result that is loaded: AC receives the result of the function and Carry the carry bit only if bits 8 and 9 are 0.

The shifter output is also tested for a skip according to the condition specified by bits 13–15. The processor skips the next instruction if the specified condition is satisfied.

Bit	*Effect of a 1 in the bit*
13	Selects the condition that the low order 16 bits of the shifter output are all 0.
14	Selects the condition that the bit in the carry position of the shifter output is 0.
15	Inverts the conditions selected by bits 13 and 14. In other words a 1 in bit 15 causes 1s in the other bits to select nonzero conditions.

The combined effects of bits 13–15 taken together and the mnemonics for the various bit configurations are as follows.

Mnemonic	*Bits 13–15*	*Skip function*
	0	Never Skip
SKP	1	Always Skip
SZC	2	Skip on Zero Carry
SNC	3	Skip on Nonzero Carry
SZR	4	Skip on Zero Result
SNR	5	Skip on Nonzero Result
SEZ	6	Skip if Either Carry or Result is Zero
SBN	7	Skip if Both Carry and Result are Nonzero

Remember that the test is made on the shifter output. Thus if the result of an addition is shifted left, SZC and SNC actually test the sign of the sum. Note also that the test is made whether or not the shifter output is loaded. The program can therefore test the result of an arithmetic function without disturbing the original operands or Carry.

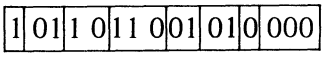
Programming Conventions. The instruction

ADD 1,2

1	01	1	0	11	0	00	00	0	000

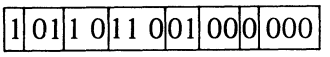
which assembles as 133000, adds the numbers in AC1 and AC2, loads the unshifted result in AC2, and complements Carry if there is a carry out of bit 0. Other carry and shift operations are selected simply by appending the appropriate letters to the function mnemonic, but the carry letter (if any) must appear first. Thus to generate a carry bit of 1 on a carry (0 otherwise) and load Carry and AC2 with the 17-bit result shifted left we give

```
ADDZL   1,2
```



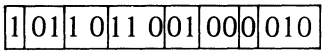
which assembles as 133120. This instruction places the sign of the sum in Carry, the rest of the sum in bits 0–14 of AC2, and a 1 or a 0 in bit 15 depending on whether or not there is a carry out of the sign bit. To use the present state of Carry instead of 0 as the basis for adjusting bit 15, but otherwise produce the same effect, give

```
ADDL    1,2
```



which assembles as 133100. The instruction

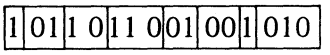
```
ADDL    1,2,SZC
```



assembles as 133102, and affects Carry and AC2 in the same manner as the preceding instruction, but also causes the processor to skip the next instruction if the sign of the sum is positive.

The initial symbol # following the expanded function mnemonic places a 1 in bit 12 to prevent the loading of the shifter output. Hence we can skip the next instruction on a positive sum without affecting AC2 or Carry by giving

```
ADDL#   1,2,SZC
```

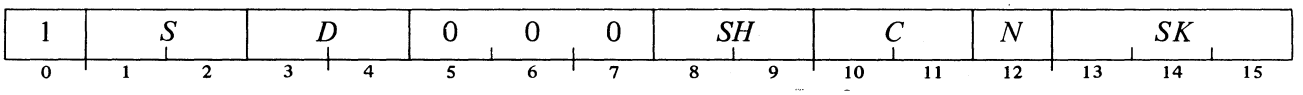


which assembles as 133112.

Arithmetic and Logical Functions

The eight functions are selected by bits 5–7 of the instruction word. For convenience the source and destination accumulators addressed by the *S* and *D* parts of the instruction are referred to as AC*S* and AC*D*.

COM **Complement** 5.6 μs



Place the (logical) complement of the word from AC*S* and place the carry bit specified by *C* in the shifter. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

Time. *M* + 3.0 μs

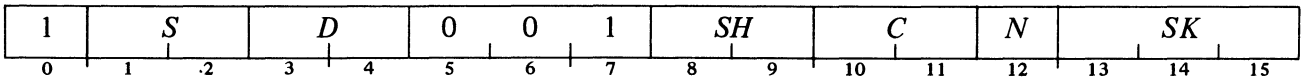
Example. Suppose we wish to test AC1 for the unsigned integer $2^{16}-1$ (177777, signed −1). The instruction

```
COM#    1,1,SZR
```

skips the next instruction if AC1 contains all 1s. The result is not loaded so we could specify any accumulator as the destination, *eg*

```
COM#    1,3,SZR
```

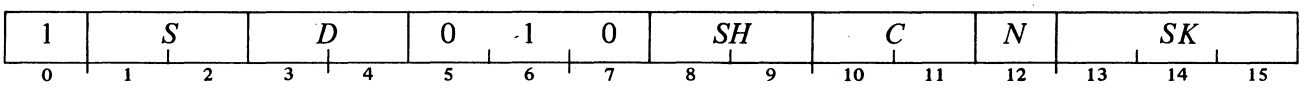
NEG **Negate** 5.6 μs



Place the twos complement of the number from AC*S* into the shifter. If AC*S* contains zero, supply the complement of the value specified by *C* as the carry bit; otherwise supply the specified value. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

Time. *M* + 3.0 μs

MOV **Move** 5.6 μs



Place the contents of AC*S* and the carry bit specified by *C* in the shifter. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

Time. *M* + 3.0 μs

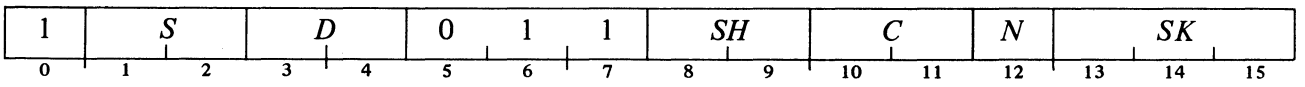
Examples. The test for a zero word in AC1 is any of these:

```
MOV   1,1,SZR      MOV   1,1,SNR      MOV#  1,1,SZR      MOV#  1,1,SNR
```

Suppose we wish to divide the number in AC2 by 2.

```
MOVL#   2,2,SZC     ;Is it positive?
MOVOR   2,2,SKP     ;No, put in a 1 and skip
MOVZR   2,2         ;Yes, put in a 0
```

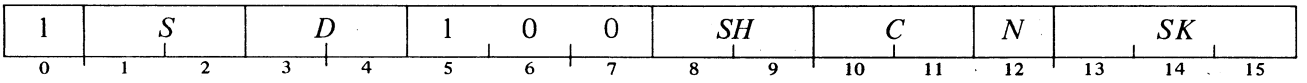
INC **Increment** 5.6 μs



Add 1 to the number from AC*S* and place the result in the shifter. If AC*S* contains $2^{16}-1$ (signed -1) supply the complement of the value specified by *C* as the carry bit; otherwise supply the specified value. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

Time. *M* + 3.0 μs

ADC **Add Complement** 5.9 μs



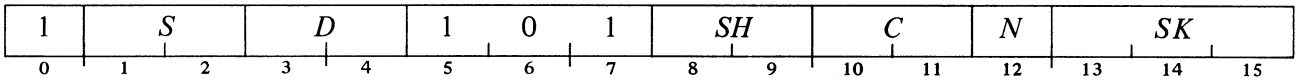
Add the (logical) complement of the number from AC*S* to the number from AC*D*, and place the result in the shifter. If AC*D* > AC*S* (unsigned), supply the complement of the value specified by *C* as the carry bit; otherwise supply the specified value. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

Time. *M* + 3.3 μs

NOTE: For signed numbers the carry condition is that the signs of the operands are the same and AC*D* is the greater, or the signs differ and AC*D* is negative.

This instruction is often used to process high order words in multiple precision subtraction, wherein a negative is usually a ones complement instead of a twos complement. The overflow condition for signed numbers using ones complement conventions is the same as that given for SUB below.

SUB **Subtract** 5.9 μs



Subtract by adding the twos complement of the number from AC*S* to the number from AC*D*, and place the result in the shifter. If AC*D*≥AC*S* (unsigned), supply the complement of the value specified by *C* as the carry bit; otherwise supply the specified value. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

Time. *M* + 3.3 μs

NOTE: For signed numbers the carry condition is that the signs of the operands are the same and AC*D*≥ AC*S*, or the signs differ and AC*D* is negative.

EXAMPLES. This instruction can be used to clear an accumulator by subtracting it from itself.

```
SUB     2,2
```

clears AC2 and complements Carry,

```
SUBO    2,2
```

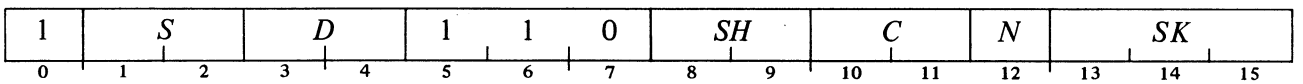
clears both AC2 and Carry.

SUB is also useful for comparing quantities, *eg*

```
SUB#    2,3,SNR
```

skips if AC2 and AC3 are unequal but does not affect either accumulator.

ADD **Add** 5.9 μs

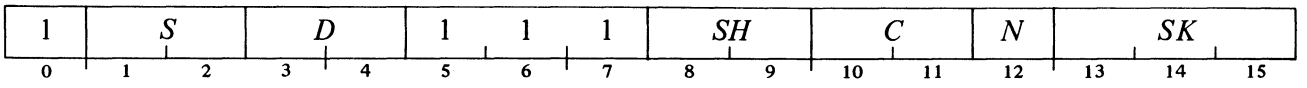


Add the number from AC*S* to the number from AC*D*, and place the result in the shifter. If the unsigned sum is $\geqslant 2^{16}$, supply the complement of the value specified by *C* as the carry bit; otherwise supply the specified value. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

Time. $M + 3.3$ μs

NOTE: For signed numbers the carry condition is that both summands are negative, or their signs differ and their magnitudes are equal or the positive one is the greater in magnitude.

AND **And** 5.9 μs



Place the logical AND function of the word from AC*S* and the word from AC*D* in the shifter. Supply the value specified by *C* as the carry bit. Perform the shift operation specified by *SH*. Load the shifter output in Carry and AC*D* unless *N* is 1. Skip the next instruction if the shifter output satisfies the condition specified by *SK*.

This instruction operates bitwise on a pair of words, so it actually performs sixteen logical operations simultaneously. A given bit of the result is 1 if the corresponding bits of both operands are 1; otherwise the resulting bit is 0.

ACS_i	ACD_i	$Result_i$
0	0	0
0	1	0
1	0	0
1	1	1

Time. $M + 3.3$ μs

Programming Examples

Together ADC and SUB allow the program to compare the magnitudes of unsigned numbers in every way. *Eg*

```
SUBZ#   1,0,SZC
```

skips if AC0<AC1, whereas,

```
ADCZ#   1,0,SZC
```

skips if AC0$\leqslant$AC1.

It is well known that the *n*th perfect square is the sum of the first *n* odd numbers. We can therefore find the largest integer contained in the square root of an integer held in AC0 by successively subtracting odd numbers in order from AC0 until overflow occurs, *ie* until AC0 becomes negative. The desired answer is the number of odd numbers successfully subtracted before a carry occurs. The routine is called by a JSR with effective address SQRT.

```
SQRT:   SUBZL   1,1         ;AC1 gets 1 (first odd number)
        MOVZL   1,2         ;AC2 gets 2 (difference between odd numbers)

Q1:     SUBZ    1,0,SZC     ;Subtract next odd number; negative yet?
        ADD     2,1,SKP     ;No, add 2 to get next odd number
        MOVZR   1,0,SKP     ;Yes, divide 2n - 1 by 2 to get n - 1
        JMP     Q1          ;Subtract again
        JMP     0,3         ;Return to call + 1
```

The instruction set has only one logical function of two variables, but the inclusive and exclusive OR functions can be performed by very simple routines. In an inclusive OR a bit of the result is 1 if either of the corresponding operand bits is 1, otherwise it is 0. The algorithm for full words is

$$A \wedge \sim B + B = A \vee B$$

Taking the arguments as single bits, if B is 1, $A \wedge \sim B$ is 0 regardless of the state of A, and the expression on the right is 1. If B is 0, the expression is 1 or 0 as A is 1 or 0. In no case are $A \wedge \sim B$ and B both 1, so the full word addition generates no carries. This sequence places the inclusive OR of AC0 and AC1 in AC1 (AC0 = B, AC1 = A).

```
COM     0,0         ;~B
AND     0,1         ;~B ∧ A in AC1
ADC     0,1         ;~~B + ~B ∧ A = B + ~B ∧ A in AC1
```

In an exclusive OR a bit of the result is 1 if the corresponding operand bits are different, otherwise it is 0. This is equivalent to the sum if carries from one bit position to the next are ignored. Now a carry out of the ith position is equal to twice the value of a 1 in the ith position, and a carry is generated only if the ith bits of both summands are 1, provided we compensate for any carry into the ith position. The algorithm is therefore.

$$A \vee B = A + B - 2(A \wedge B)$$

This sequence places the exclusive OR of AC0 and AC1 in AC1, destroying the contents of AC2 and Carry in the process (AC0 = B,AC1 = A).

```
MOV     1,2         ;Move A to AC2
ANDZL   0,2         ;2(A ∧ B) in AC2
ADD     0,1         ;A + B
SUB     2,1         ;A + B - 2(A ∧ B)
```

Double Precision Arithmetic. A double length number consists of two words concatenated into a 32-bit string wherein bit 0 is the sign and bits 1–31 are the magnitude in twos complement notation. The high order part of a negative number is therefore in ones complement form unless the low order part is null (at the right

$+262{,}146_{10} = +2000002_8 =$

0 000 000 000 001 000	0 000 000 000 000 010
0 15	16 31

$-262{,}146_{10} = -2000002_8 =$

1 111 111 111 110 111	1 111 111 111 111 110
0 15	16 31

only 0s are null regardless of sign). Hence in processing double length numbers, twos complement operations are usually confined to the low order parts, whereas ones complement operations are generally required for the high order parts.

Suppose we wish to negate the double length number whose high and low order words respectively are in AC0 and AC1. We negate the low order part, but we simply complement the high order part unless the low order part is zero. Hence

```
NEG     1,1,SNR
NEG     0,0,SKP     ;Low order zero
COM     0,0         ;Low order nonzero
```

Note that the magnitude parts of the sequence of negative numbers from the most negative toward zero are the positive numbers from zero upward. In other words the negative representation $-x$ is the sum of x and the most negative number. Hence in multiple precision arithmetic, low order words can be treated simply as positive numbers. In unsigned addition a carry indicates that the low order result is just too large and the high order part must be increased. We add the number in AC2 and AC3 to the number in AC0 and AC1.

```
ADDZ    3,1,SZC
INC     2,2
ADD     2,0
```

In twos complement subtraction a carry should occur unless the subtrahend is too large. We could increment as in addition, but since incrementing in the high order part is precisely the difference between a ones complement and a twos complement, we can always manage with only two instructions. We subtract the number in AC2 and AC3 from that in AC0 and AC1.

```
SUBZ    3,1,SZC
SUB     2,0,SKP
ADC     2,0
```

Multiply and Divide Subroutines. In pencil and paper decimal multiplication, one multiplies the multiplicand by each multiplier digit separately to form a set of partial products. Successive partial products are shifted one place to the left (they are multiplied by successive powers of 10) and summed. In the computer it is easier to add each partial product as it is formed and shift the result one place to the right so the running sum is in the correct position to receive the next one. Since the numbers are binary, each partial product is either the multiplicand or zero. Hence at each step we either add the multiplicand and shift or simply shift depending on whether the next bit of the multiplier is 1 or 0.

The multiply subroutine operates on unsigned integers in AC1 and AC2 to generate a double length product whose high and low order parts are left in AC0 and AC1 respectively. If entry is made at MULT0, the product is added to the number originally in AC0 (the result is AC0 + AC1 × AC2). Carry is left unchanged.

```
MULT:   SUBC    0,0         ;Clear AC0, don't disturb Carry
MULT0:  STA     3,MSAV      ;Save return
        LDA     3,M20       ;Get count

MLOOP:  MOVR    1,1,SNC     ;Check next multiplier bit
        MOVR    0,0,SKP     ;0 – shift
```

```
        ADDZR   2,0             ;1 – add multiplicand and shift
        INC     3,3,SZR         ;Count step, complementing Carry on final count
        JMP     MLOOP
        MOVCR   1,1             ;Shift in last low bit (which was complemented by final count)
        JMP     @MSAV           ;and restore Carry

M20:    –20                     ;16₁₀ steps
MSAV:   0
```

The divide subroutine also operates on unsigned integers, using a double length dividend and a single length divisor to produce a single length quotient and remainder. The routine starts by comparing the divisor with the high order half of the dividend: if the divisor is less than or equal to the latter quantity, the division is not performed as the result would be greater than $2^{16}-1$, the largest integer than can be held in an accumulator. (The result would be greater than or equal to 1 if the operands are interpreted as proper fractions.) It is not a sensible procedure simply to compute the first sixteen bits of the quotient as it would be impossible to determine the order of magnitude. So it is up to the programmer to shift the dividend to the correct position beforehand. For operations limited to single length integers (referred to as "integer division") the one-word dividend is treated as the low order half of a double length number whose high order part is null, and the routine fails to perform the division only when the divisor is zero. The worst possible case is the division of $2^{16}-1$ by 1, whose integral result can be accommodated.

In division on paper, one subtracts out the divisor the number of times it goes into the dividend, then shifts the dividend one place to the left (or the divisor to the right) and again subtracts out. In binary computations the divisor goes into the dividend either once or not at all. Each comparison thus generates a single bit of the quotient. If the divisor does go in, it is subtracted and a 1 is entered into the quotient; if not, a 0 is entered. The test condition is reversed if the dividend shift puts a 1 in Carry; this way Carry is used effectively as an extra magnitude bit and no information is lost in the shift.

The high and low parts of the dividend are in AC0 and AC1, the divisor is in AC2. At completion the remainder is in AC0, the quotient is in AC1, and AC2 is unchanged.

```
        JSR     DIV
        ...             ;Return here if division not performed
        ...             ;Normal return
```

For integer division entry is at DIV0 with the dividend in AC1. If the division is not performed the three accumulators are unchanged except that calling DIV0 clears AC0. Note that the result is such that if MUL0 is called, AC0 and AC1 are restored, *ie* divisor times quotient plus remainder equals original dividend.

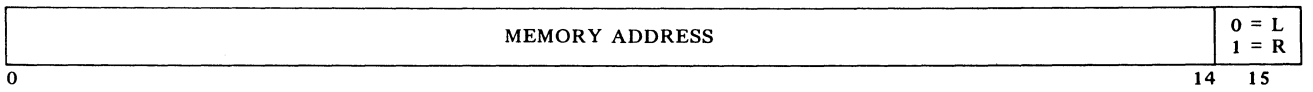
```
DIV0:   SUB     0,0             ;Integer divide – clear high part

DIV:    SUBZ#   2,0,SZC         ;Test for overflow
        JMP     0,3             ;Yes, exit (AC0≥AC2)
        STA     3,MSAV          ;Save return (see multiply)
        LDA     3,M20           ;Get step count (see multiply)
        MOVZL   1,1             ;Shift dividend low part

DLOOP:  MOVL    0,0             ;Shift dividend high part
        SUB#    2,0,SZC         ;Does divisor go in?
        SUB     2,0             ;Yes
```

```
        MOVL    1,1             ;Shift dividend low part
        INC     3,3,SZR         ;Count step
        JMP     DLOOP
        ISZ     MSAV            ;Increment return
        JMP     @MSAV           ;Return
```

Byte Manipulation. For processing 8-bit bytes it is convenient to use a byte pointer in which bits 0–14 are the address of the memory location that contains or will receive the byte, and bit 15 specifies which half (0 left,



1 right). Incrementing a pointer with this format changes bit 15 every count to specify the next byte, but changes the address part only every other count.

The following subroutine picks up a byte, places it in the right half of AC0, and increments the byte pointer in memory. The calling sequence is

```
        JSR     PICK
        . . .                   ;Address of pointer
        . . .                   ;Return here if byte is zero
        . . .                   ;Normal return
```

The calling sequence supplies the address of the location containing the pointer. A separate return for a zero byte allows the program to process a sequence of bytes whose length is unspecified, but which terminates with a zero byte.

```
PICK:   LDA     2,@0,3          ;Get byte pointer
        ISZ     @0,3            ;Increment pointer
        MOVZR   2,2             ;Put address in right place (left/right bit to Carry)
        LDA     0,0,2           ;Bring memory word to AC0
        LDA     2,C377          ;Get 8-bit mask
        MOV     0,0,SNC         ;Test Carry for which half
        MOVS    0,0             ;Swap byte from left to right
        AND     2,0,SNR         ;Mask out unwanted byte and test for zero
        JMP     1,3             ;Zero, return to call + 2
        JMP     2,3             ;Nonzero, return to call +3
C377:   377                     ;8-bit mask (1s in right half)
```

2.3 INPUT-OUTPUT

Instructions in the in-out class govern all transfers of data to and from the peripheral equipment, and also perform various operations within the processor. An instruction in this class is designated by 011 in bits 0–2. Bits 10–15 select the device that is to respond to the instruction. The format thus allows for 64 codes of which 62 can be used to address devices (octal 01–76). The code 00 is not used, and 77 is used for a number of special functions including reading the console data switches and controlling the program interrupt. A table in

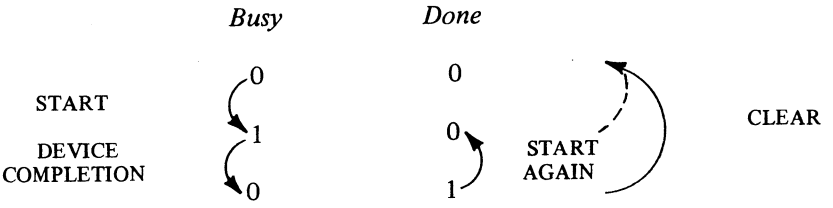
Appendix E lists all devices for which codes have been assigned, and gives their mnemonics and DGC option numbers.

Every device has a 6-bit device selection network, an Interrupt Disable flag, and Busy and Done flags. The selection network decodes bits 10–15 of the instruction so that only the addressed device responds to signals sent by the processor over the in-out bus. The Busy and Done flags together denote the basic state of the device. When both are clear the device is idle. To place the device in operation, the program sets Busy. If the device will be used for output, the program must give a data-out instruction that sends the first unit of data — a word or character depending on how the device handles information. (The word "output" used without qualification always refers to the transfer of data from the processor to the peripheral equipment; "input" refers to the transfer in the opposite direction.) When the device has processed a unit of data, it clears Busy and sets Done to indicate that it is ready to receive new data for output, or that it has data ready for input. In the former case the program would respond with a data-out instruction to send more data; in the latter with a data-in instruction to bring in the data that is ready. If the Interrupt Disable flag is clear, the setting of Done signals the program by requesting an interrupt; if the program has set Interrupt Disable, then it must keep testing Done or Busy to determine when the device is ready.

In all in-out instructions bits 8 and 9 either control or sense Busy and Done. In those instructions in which bits 8 and 9 specify a control function, the mnemonics and bit configurations and the functions they select are as follows.

Mnemonic	*Bits 8–9*	*Control function*
	00	None
S	01	Start the device by clearing Done and setting Busy
C	10	Clear both Busy and Done, idling the device
P	11	Pulse the special in-out bus control line — the effect, if any, depends on the device

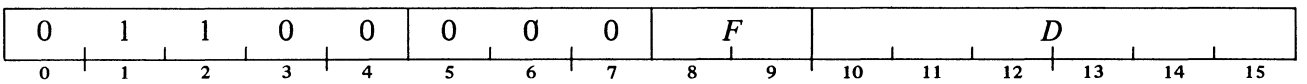
The overall sequence of Busy and Done states is determined by both the program and the internal operation of the device.



The data-in or data-out instruction that the program gives in response to the setting of Done can also restart the device. When all the data has been transferred the program generally clears Done so the device neither requests further interrupts nor appears to be in use, but this is not necessary. Busy and Done both set is a meaningless situation.

Bits 5–9 specify the complete function to be performed. If there is no transfer (bits 5–7 all alike), bits 3 and 4 are ignored and bits 8 and 9 may specify a control function or a skip condition.

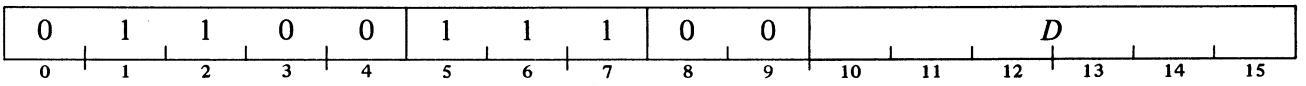
NIO **No IO Transfer** 4.4 μs



Perform the control function specified by *F* in device *D*.

Time. *M* + 1.8 μs

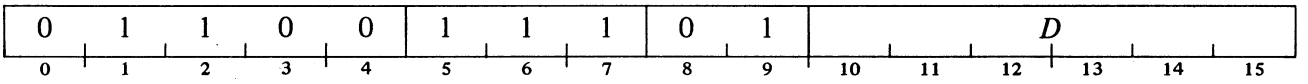
SKPBN **Skip if Busy is Nonzero** 4.4 μs



Skip the next instruction in sequence if the Busy flag in device *D* is 1.

Time. *M* + 1.8 μs

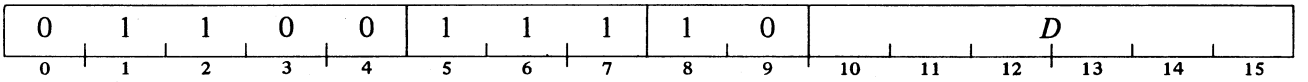
SKPBZ **Skip if Busy is Zero** 4.4 μs



Skip the next instruction in sequence if the Busy flag in device *D* is 0.

Time. *M* + 1.8 μs

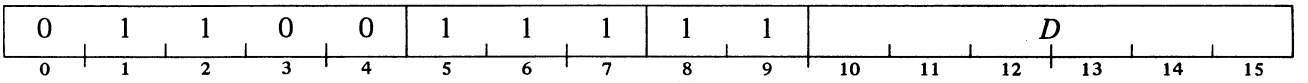
SKPDN **Skip if Done is Nonzero** 4.4 μs



Skip the next instruction in sequence if the Done flag in device *D* is 1.

Time. *M* + 1.8 μs

SKPDZ **Skip if Done is Zero** 4.4 μs



Skip the next instruction in sequence if the Done flag in device *D* is 0.

Time. *M* + 1.8 μs

The letter for the control function is appended to the basic mnemonic; NIO alone with any device code is a no-op. To place say the high speed tape reader in operation we could give

```
NIOS    12
```

which assembles as 060112 (0 110 000 001 001 010) and causes the reader to read one line from tape into its buffer. There are mnemonics for the device codes so we could also give the equivalent

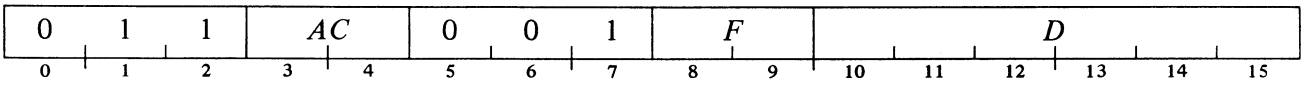
```
NIOS    PTR
```

To determine when the character is in the buffer without using the program interrupt we can wait for either Busy to clear or Done to set, *eg* by giving

```
SKPDN   PTR
JMP     .-1
```

If bits 5–7 are not all alike the instruction calls for an in-out transfer. Bits 3 and 4 specify the accumulator that supplies or receives the data, bits 8 and 9 specify a control function (if any) as listed above.

DIA **Data In A** 4.4 μs

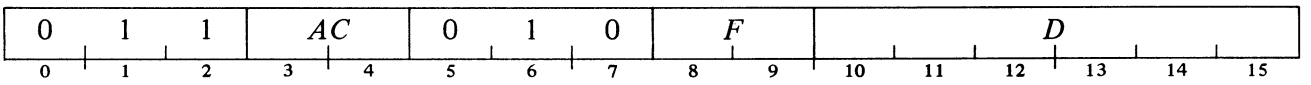


Move the contents of the A buffer in device *D* to accumulator *AC*, and perform the function specified by *F* in device *D*.

The number of data bits moved depends on the size of the device buffer, its mode of operation, etc. Bits in *AC* that do not receive data are cleared.

Time. *M* + 1.8 μs

DOA **Data Out A** 4.7 μs

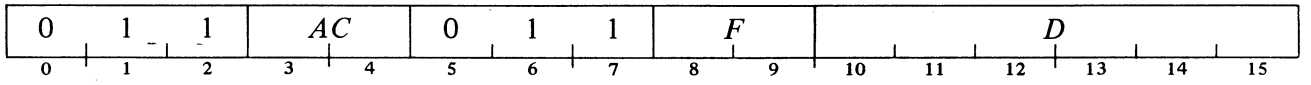


Send the contents of accumulator *AC* to the A buffer in device *D*, and perform the function specified by *F* in device *D*.

The amount of data actually accepted by the device depends on the size of its buffer, its mode of operation, etc. The original contents of *AC* are unaffected.

Time. *M* + 2.1 μs

DIB **Data in B** 4.4 μs

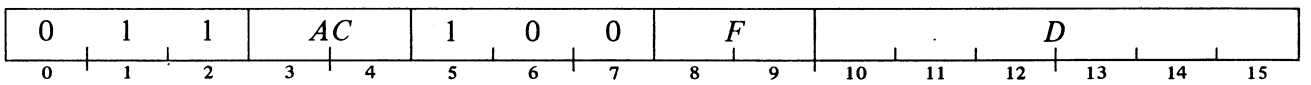


Move the contents of the B buffer in device *D* to accumulator *AC*, and perform the function specified by *F* in device *D*.

The number of data bits moved depends on the size of the device buffer, its mode of operation, etc. Bits in *AC* that do not receive data are cleared.

Time. *M* + 1.8 μs

DOB **Data Out B** 4.7 μs

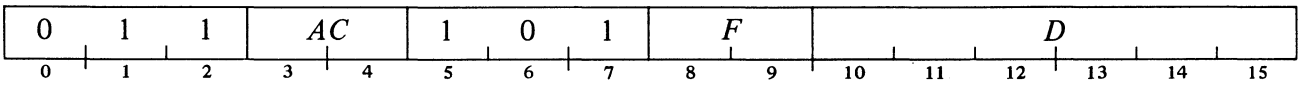


Send the contents of accumulator *AC* to the B buffer in device *D*, and perform the function specified by *F* in device *D*.

The amount of data actually accepted by the device depends on the size its buffer, its mode of operation, etc. The original contents of *AC* are unaffected.

Time. $M + 2.1$ μs

DIC **Data in C** 4.4 μs

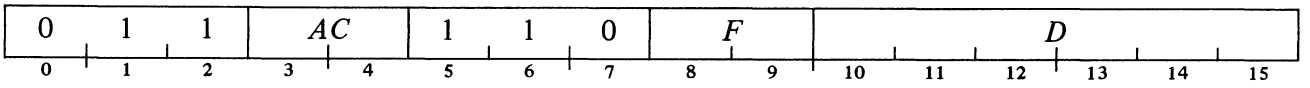


Move the contents of the C buffer in device *D* to accumulator *AC*, and perform the function specified by *F* in device *D*.

The number of data bits moved depends on the size of the device buffer, its mode of operation, etc. Bits in *AC* that do not receive data are cleared.

Time. $M + 1.8$ μs

DOC **Data Out C** 4.7 μs



Send the contents of accumulator *AC* to the C buffer in device *D*, and perform the function specified by *F* in device D.

The amount of data actually accepted by the device depends on the size of its buffer, its mode of operation, etc. The original contents of *AC* are unaffected.

Time. $M + 2.1$ μs

A device may require no IO transfers, such as the real time clock, which uses only NIOS and NIOC to turn it on and off. All of the simpler data handling devices have only an A buffer, *eg* to hold a single character in the teletypewriter, tape reader and tape punch, or to receive incremental plotting data for a single point in the plotter. Suppose the reader has read a line from tape into its buffer. We can bring the character into the right half of AC2 by giving

```
DIA     2,PTR
```

If we want to read another line we can make the transfer with a

```
DIAS    2,PTR
```

which brings the character into AC2, clears Busy, and sets Done causing the reader to read the next line. If the buffer contains the final character to be read from tape we might give

```
DIAC    2,PTR
```

which retrieves the character and clears Done. Data is moved in and out in characters of various sizes or in full 16-bit words. Generally a device uses only DIA and/or DOA for data but it may use other IO transfer instructions to handle status and control information. A high speed device, such as magnetic tape or disk, may

require IO transfer instructions *only* for status and control information with data moving directly between the device and memory via the data channel.

Most peripheral devices involve motion of some sort, usually mechanical. In this respect there are two types of devices, those that stay in motion and those that do not. Magnetic tape is an example of the former type. Here the device executes a command (such as read, write, space forward) and Done sets when the entire operation is finished. A separate flag requests a data channel transfer each time the device is ready for direct data access to memory, but the tape keeps moving until an entire record or file has been processed. Paper tape, on the other hand, stops after each line is read, but if the program gives another DIAS within a critical time the tape moves continuously.

Other devices operate in one or the other of these two ways but differ in various respects. The tape punch and teletype output are like the reader. Teletype input is initiated by the operator striking a key rather than by the program. Once started the card reader reads an entire card, with a DIA required for each column.

In the remainder of this manual the discussion of each device treats only the control functions and the applicable IO transfer instructions. The skips apply to all and are the same in all cases. Giving a data-in instruction that does not apply to a device (either because the device is output only or does not have the buffer specified) clears the addressed accumulator but does do the specified control function. Similarly a data-out that does not apply is a no-op except for control functions. When the device code is undefined or the addressed device is not in the system, any data-out, an SKPBN or an SKPDN is a no-op, an SKPBZ or SKPDZ is an absolute skip, and any data-in simply clears the addressed AC.

All instructions discussed in the rest of this manual are in-out instructions with various device codes. For the transfer instructions the mnemonics are given with a dash in the position occupied by an accumulator address, as the assembler indicates an error if the programmer fails to specify an accumulator. The programmer must substitute a valid address for the dash. In the format box for each instruction the accumulator address part is represented by *AC*. In the instruction description, "AC" refers to the accumulator specified by the *AC* part of the instruction word.

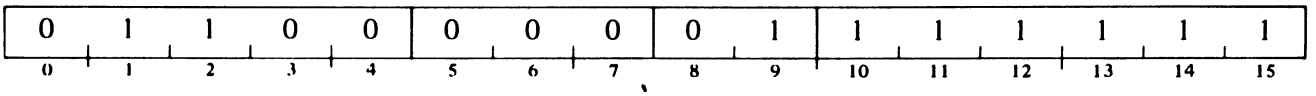
Special Code-77 Functions

In-out instructions with the code 77 in bits 10–15 perform a number of special functions rather than controlling a specific device. In all but the skip instructions bits 8 and 9 are used to turn the interrupt on and off. The mnemonics are the same as those for controlling Busy and Done in IO devices, but with code 77 they select the following special functions.

Mnemonic	*Function*
S	Set the Interrupt On flag to enable the processor to respond to interrupt requests.
C	Clear the Interrupt On flag to prevent the processor from responding to interrupt requests.
P	None

Most of these instructions perform functions associated with processor elements so the mnemonic for 77 is CPU. For the transfer type instructions that use no accumulator, the mnemonics are given with an accumulator address included, as the assembler indicates an error if the programmer fails to specify an accumulator even when none is used. A zero address is given, but any valid address would suffice. Instructions for the program interrupt and power failure detection are treated in greater detail in later sections.

NIOS CPU **Interrupt Enable** 4.4 μs

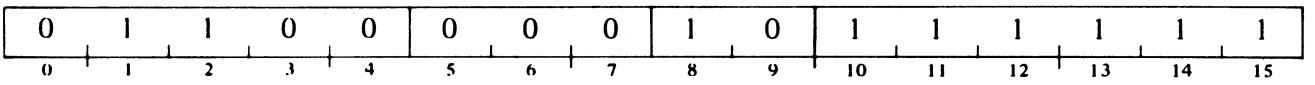


Set the Interrupt On flag to allow the processor to respond to interrupt requests.

Time. $M + 1.8$ μs

NOTE: The assembler recognizes the mnemonic INTEN as equivalent to NIOS CPU.

NIOC CPU **Interrupt Disable** 4.4 μs

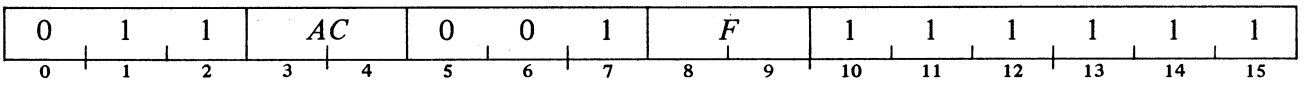


Clear the Interrupt On flag to prevent the processor from responding to interrupt requests.

Time. $M + 1.8$ μs

NOTE: The assembler recognizes the mnemonic INTDS as equivalent to NIOC CPU.

DIA –,CPU **Read Switches** 4.4 μs

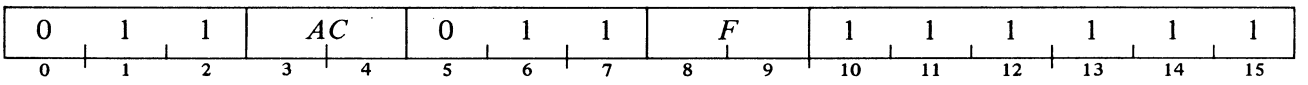


Read the contents of the console data switches into AC, and perform the function specified by *F*.

Time. $M + 1.8$ μs

NOTE: The assembler recognizes the mnemonic READS as equivalent to DIA –,CPU.

DIB –,CPU **Interrupt Acknowledge** 4.4 μs

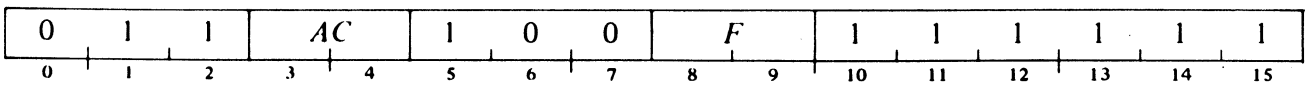


Place in AC bits 10–15 the device code of the first device on the bus that is requesting an interrupt ("first" means the one that is physically closest to the processor on the bus). Perform the function specified by *F*.

Time. $M + 1.8$ μs

NOTE: The assembler recognizes the mnemonic INTA as equivalent to DIB –,CPU.

DOB –,CPU **Mask Out** 4.7 μs

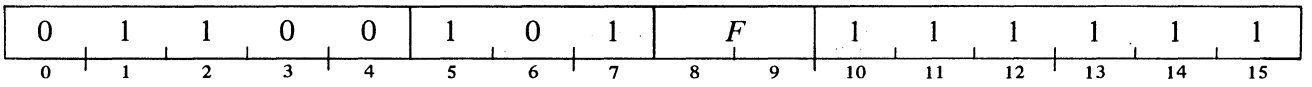


Set up the Interrupt Disable flags in the devices according to the mask in AC. For this purpose each device is connected to a given data line, and its flag is set or cleared as the corresponding bit in the mask is 1 or 0. Perform the function specified by *F*.

Time. *M* + 2.1 μs

NOTE: The assembler recognizes the mnemonic MSKO as equivalent to DOB –,CPU.

DIC 0,CPU **Clear IO Devices** 4.4 μs

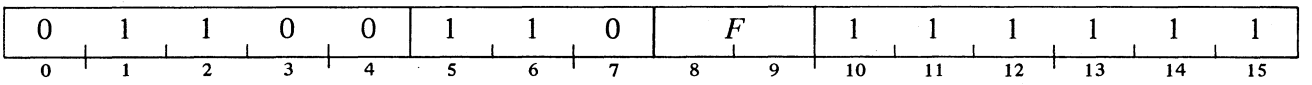


Clear the control flipflops, including Busy, Done and Interrupt Disable, in all devices connected to the bus. Perform the function specified by *F*.

Time. *M* + 1.8 μs

NOTE: The assembler recognizes the mnemonic IORST as equivalent to DICC 0,CPU— *ie as* the instruction defined here with *F* set to 10.

DOC 0,CPU **Halt** 4.7 μs

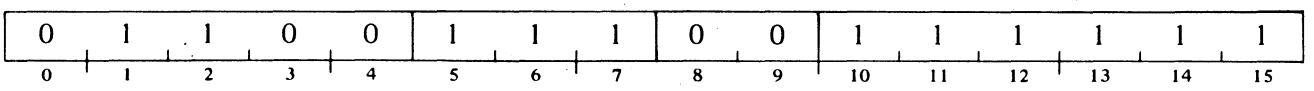


Perform the function specified by *F* and then halt the processor. When the processor stops, the instruction and data lights display the halt instruction, the address lights point to the location following the halt instruction.

Time. *M* + 2.1 μs

NOTE: The assembler recognizes the mnemonic HALT as equivalent to DOC 0,CPU.

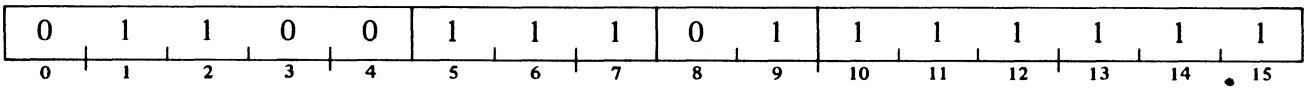
SKPBN CPU **Skip if Interrupt On is Nonzero** 4.4 μs



Skip the next instruction in sequence if the Interrupt On flag is 1.

Time. *M* + 1.8 μs

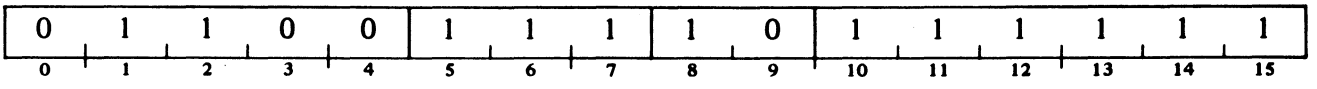
SKPBZ CPU **Skip if Interrupt On is Zero** 4.4 μs



Skip the next instruction in sequence if the Interrupt On flag is 0.

Time. *M* + 1.8 μs

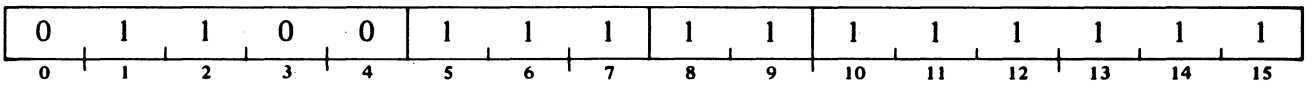
SKPDN CPU **Skip if Power Failure is Nonzero** **4.4 μs**



Skip the next instruction in sequence if the Power Failure flag is 1.

Time. $M + 1.8$ μs

SKPDZ CPU **Skip if Power Failure is Zero** **4.4 μs**



Skip the next instruction in sequence if the Power Failure flag is 0.

Time. $M + 1.8$ μs

The assembler recognizes a number of convenient mnemonics for instructions with device code 77.

Mnemonic	*Meaning*	*Mnemonic Equivalent*		*Octal Equivalent*
READS	Read Switches	DIA	–,CPU	060477
IORST	IO Reset	DICC	0,CPU	062677
HALT	Halt	DOC	0,CPU	063077
INTENT	Interrupt Enable	NIOS	CPU	060177
INTDS	Interrupt Disable	NIOC	CPU	060277
INTA	Interrupt Acknowledge	DIB	–,CPU	061477
MSKO	Mask Out	DOB	–,CPU	062077

Eg to read the switches into AC3 we could simply give

```
READS   3
```

instead of

```
DIA     3,CPU
```

However, there is one important difference between these special mnemonics and the standard ones: mnemonics for turning the interrupt on and off cannot be appended to them! Thus to set Interrupt On while reading the

switches we must give

```
DIAS    3,CPU
```

Note that IORST clears Interrupt On along with the devices on the bus. We can set it while clearing the devices by giving

```
DICS    0,CPU
```

2.4 PROGRAM INTERRUPT

Many in-out devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them, but they must be serviced within a short time after they request it. Failure to service within the specified time (which varies among devices) can often result in loss of information and certainly results in operating the device below its maximum speed. The program interrupt is designed with these considerations in mind, *ie* the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices. The hardware also allows conditions internal to the processor to signal the program by requesting an interrupt.

Interrupt Requests. Interrupt requests by a device are governed by its Done and Interrupt Disable flags. When a device completes an operation it sets Done, and this action requests a program interrupt if Interrupt Disable is clear — if Interrupt Disable has been set by the program the device cannot request an interrupt. At the beginning of every memory cycle the processor synchronizes any requests that are then being made. Once a request has been synchronized the device that made it must wait for an interrupt to start. The request signal is a level so once synchronized it remains on the bus until the program clears Done or sets Interrupt Disable. If the program does set the Interrupt Disable flag in a device, that device not only cannot request an interrupt when its Done flag sets, but any request it has already made and had synchronized is disabled, so it is no longer waiting for an interrupt.

Starting an Interrupt. The processor starts an interrupt if all four of the following conditions hold:

- The processor had just completed an instruction or a data channel transfer [see §2.5].
- At least one device is waiting for an interrupt to start (*ie* it was requesting an interrupt at the beginning of the last memory cycle).
- Interrupts are enabled, *ie* Interrupt On is set.
- No device is waiting for a data channel transfer, *ie* there are no data channel requests that the processor has synchronized but not yet fulfilled. The data channel has priority over program interrupts.

When the processor finishes an instruction it takes care of all data channel requests before it starts an interrupt; this includes any additional data channel requests that are synchronized while data channel transfers are being made. When no more devices are waiting for data channel transfers, the processor starts an interrupt if Interrupt On is set and a device was requesting an interrupt at the beginning of the last data channel transfer.

The processor starts an interrupt by clearing Interrupt On so no further interrupts can be started, saving PC (which points to the next instruction) in location 0, and simulating a JMP @1 to jump to the interrupt service routine. Location 1 should contain the address of the routine or an indirect address that will get there.

Servicing an Interrupt. The interrupt service routine should determine which device requires service, save the contents of any accumulators that will be used in the routine, save Carry if it will be used, and service the device. The routine can identify the device by testing with IO skips or by giving an interrupt acknowledge instruction (INTA). This instruction determines which is the first device on the bus that is waiting for service by reading its device code into an accumulator. The program can simply leave the interrupt off while servicing

the device (by leaving Interrupt On clear), or it can enable interrupts and establish a priority structure that allows higher priority devices to interrupt the current device service routine. This priority is determined by a mask that controls the states of the Interrupt Disable flags in the various devices. If this final course is taken the routine must save location 0, so the return address to the interrupted program will not be lost should another interrupt occur.

Device Priority. There are several ways in which priorities are determined for or assigned to devices on the bus. An elementary priority is established by the hardware for devices that are requesting interrupts simultaneously in that the interrupt acknowledge instruction reads the code of one and only one device: among those that are waiting it reads the code of that one which is physically closest to the processor on the bus. This however applies only to those devices that are waiting at the time the acknowledgement is given. Using IO skips to determine which device to service establishes a priority by the order in which the devices are tested, but again this applies only to those that are waiting at the time.

The most significant method is by specifying which devices can interrupt a service routine currently in progress. This is done through the use of a mask that sets up the Interrupt Disable flags. Every device is wired to a particular data line on the bus and hence to a particular bit of the mask. Although slower devices are assigned to the higher numbered bits in the mask, there is no established priority as the program can use any mask configuration. All devices whose Interrupt Disable flags are set cannot cause an interrupt to start (setting Interrupt Disable causes the withdrawal of any request that has already been made and prevents the setting of Done from making a request) and are therefore regarded by the program as being of lower priority. Those devices in which Interrupt Disable is left clear can interrupt the current routine and therefore are regarded by the program as being of higher priority.

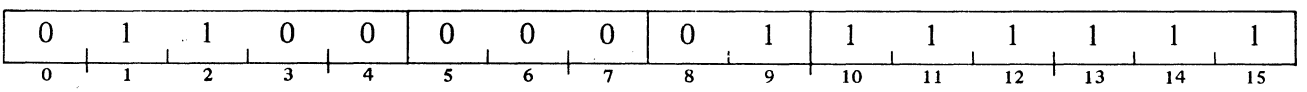
By means of the mask the program can establish any priority structure with one limitation: in some cases two or more devices are assigned to the same bit in the mask and are thus all at the same priority level. When an interrupt is in progress for a device, the rest of the devices assigned to the same mask bit must be regarded as all of lower priority or all of higher priority depending upon whether they are disabled or not.

Dismissing an Interrupt. After servicing a device the routine should restore the pre-interrupt states of the accumulators and Carry, turn on the interrupt, and jump to the interrupted program. The instruction that enables the interrupt sets Interrupt On, but the flag has no effect until the next instruction begins. Thus after the instruction that turns the interrupt back on, the processor always executes one more instruction (assumed to be the return to the interrupted program) before another interrupt can start.

If the service routine allows interrupts by higher priority devices, then before dismissing as indicated above, the routine should turn off the interrupt to prevent further interrupts during dismissal. In dismissing, the routine should reenable lower priority devices that were not allowed to interrupt the current routine but will be allowed to interrupt the program to which the processor is returning.

Instructions. The instructions for the program interrupt use special device code 77. Bits 8 and 9 of the skip instructions sense whether the interrupt is on or off; in the other instructions these bits turn the interrupt on or off by setting or clearing the Interrupt On flag (these are respectively the start and clear IO control functions).

NIOS CPU **Interrupt Enable** 4.4 μs



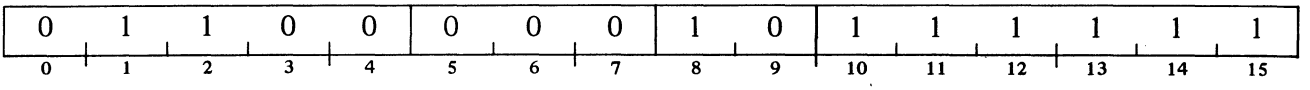
Set Interrupt On to allow the processor to respond to interrupt requests. If Interrupt On actually changes state

(0 → 1) the processor will execute one more instruction before it can start an interrupt.

Time. $M + 1.8$ μs

NOTE: The assembler recognizes the mnemonic INTEN as equivalent to NIOS CPU.

NIOC CPU **Interrupt Disable** 4.4 μs

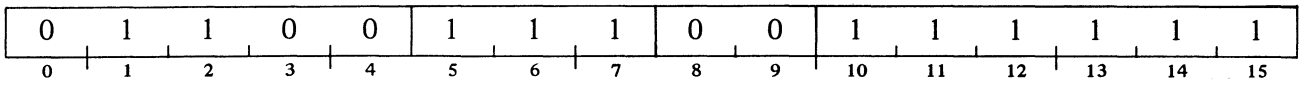


Clear Interrupt On to prevent the processor from responding to interrupt requests.

Time. $M + 1.8$ μs

NOTE: The assembler recognizes the mnemonic INTDS as equivalent to NIOC CPU.

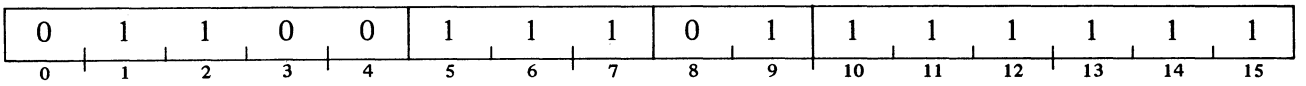
SKPBN CPU **Skip if Interrupt On is Nonzero** 4.4 μs



Skip the next instruction in sequence if Interrupt On is 1.

Time. $M + 1.8$ μs

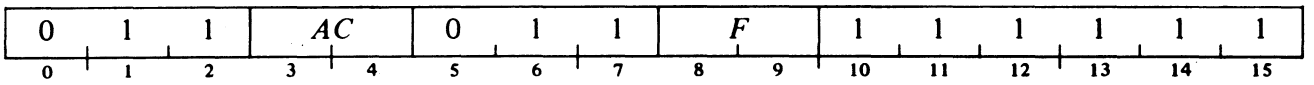
SKPBZ CPU **Skip if Interrupt On is Zero** 4.4 μs



Skip the next instruction in sequence if Interrupt On is 0.

Time. M + 1.8 μs

DIB –,CPU **Interrupt Acknowledge** 4.4 μs

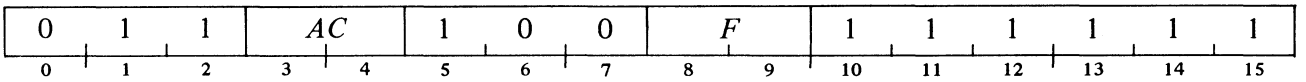


Place in AC bits 10–15 the device code of the first device on the bus that is requesting an interrupt, and perform the function specified by *F*.

Time. $M + 1.8$ μs

NOTE: The assembler recognizes the mnemonic INTA as equivalent to DIB –,CPU.

DOB –,CPU **Mask Out** 4.7 μs



Set up the Interrupt Disable flags in the devices according to the mask in AC (a 1 in a mask bit sets the flags in all devices assigned to that bit; a 0 clears them). Perform the function specified by *F*.

The following lists the devices assigned to the bits in the mask, and for each bit gives the mask for disabling all devices assigned to that and all higher numbered bits. [*Complete information on all devices is given in Appendix E.*]

AC Bit	*Devices*	*Mask*
0		177777
1		77777
2		37777
3		17777
4	Incremental IBM tape	7777
5	Line printer	3777
6		1777
7	2400/4800 baud data phone	777
8		377
9		177
10	Card reader	77
11	Paper tape reader	37
12	Plotter	17
13	Real time clock, paper tape punch, display	7
14	Teletype in	3
15	Teletype out	1

A zero mask clears all Interrupt Disable flags. In general the devices are in order by speed, with the fastest ones (those requiring the quickest service) assigned to the lower numbered bits.

Time. *M* + 2.1 μs

NOTE: The assembler recognizes the mnemonic MSKO as equivalent to DOB –,CPU.

The assembler recognizes special mnemonics for some of the above instructions.

INTEN	NIOS	CPU	Interrupt Enable	060177
INTDS	NIOC	CPU	Interrupt Disable	060277
INTA	DIB	–,CPU	Interrupt Acknowledge	061477
MSKO	DOB	–,CPU	Mask Out	062077

To turn the interrupt on or off while acknowledging or masking, the programmer must use the DIB and DOB forms – the S and C mnemonics cannot be appended to INTA and MSKO.

Timing. The time a device must wait for an interrupt to start depends on how many devices are using interrupts, how long the services routines are for devices of higher priority, and whether the data channel is in use. A single device will shut out all others of lower priority if every time its service routine dismisses the

interrupt, it is already waiting with another request; and the data channel shuts out all interrupts when it operates at the maximum rate. If the data channel is not in use and only one device is using interrupts, it need never wait longer than the time required for the processor to finish the instruction that is being performed when the request is made. The maximum instruction time for a 4K memory is about 6 μs with direct addressing, but an additional 2.6 μs is required for every level of indirect addressing.

The time the processor takes to start an interrupt is two processor cycles to store PC in location 0 and retrieve the address from location 1. With a 4K memory this is 5.2 μs if location 1 contains a direct address. The time in general is

$$2M_0 + \Sigma M_{ind} \quad \mu s$$

where M_0 is the processor cycle time for the memory that contains locations 0 and 1, and the term on the right represents the time for additional levels of indirect addressing if location 1 contains an indirect address.

Sample Master Interrupt Routine. Suppose we are using only the teletype and the high speed reader and punch. We shall allow higher priority devices to interrupt a lower priority service routine; but since the reader is the highest priority device, we shall simply leave the interrupt off while servicing it. Because of the small number of devices we can use flag testing to identify the one that is requesting service and we can treat the teletype input and output as the same priority. For illustration let us assume that the reader and punch routines use all the accumulators but the teletype routines use only AC0.

```
.LOC    0                ;This pseudoinstruction causes the assembler to put the next statement in
                         ; the location specified
        0                ;Clear location 0 -- will be used for saving PC
        INTRP            ;Put address of master interrupt processor routine in location 1
CMASK:  0                ;Will save current mask here (initially zero)
        .
        .
        .

;When the processor is interrupted the interrupt is disabled and there is an automatic jump to INTRP.
;First find source of interrupt.

INTRP:  SKPDZ   PTR      ;Try reader first
        JMP     PTRIN    ;Yes, service it
        SKPDZ   PTP      ;No, try punch
        JMP     PTPIN    ;Jump to punch service

        STA     0,TTSAV  ;Neither, must be teletype; save AC0
        LDA     0,0      ;Save return address from location 0
        STA     0,TTSAV+1
        LDA     0,CMASK  ;Save current mask
        STA     0,TTSAV+2
        LDA     0,CN3    ;Set mask bits 14, 15 (disable teletype interrupts)
        STA     0,CMASK  ;Set new current mask
        DOBS    0,CPU    ;MSKO and enable interrupts

        SKPDZ   TTO      ;Test teletype output
        JMP     TTOIN    ;Jump to output service
        SKPDN   TTI      ;Test input
        JMP     ERROR    ;Something wrong – nobody wants service
```

```
                .                               ;Service teletype in
                .
                .
                JMP     TTDSM                   ;Must dismiss

TTOIN:          .                               ;Service teletype out
                .
                .

TTDSM:  INTDS                                   ;To dismiss, first disable interrupts
                LDA     0,TTSAV+2               ;Restore previous mask
                STA     0,CMASK
                MSKO    0
                LDA     0,TTSAV                 ;Restore AC0
                INTEN                           ;Enable interrupts
                JMP     @TTSAV+1                ;Return to interrupted program

TTSAV:  0                                       ;Save AC0 here
                0                               ;Save PC (from location 0) here
                0                               ;Save current mask here
CN3:    3
;Punch routine
PTPIN:  STA     0,PPSAV                 ;Save accumulators
                STA     1,PPSAV+1
                STA     2,PPSAV+2
                STA     3,PPSAV+3
                MOVL    0,0             ;Save Carry
                STA     0,PPSAV+4
                LDA     0,0             ;Save location 0
                STA     0,PPSAV+5
                LDA     0,CMASK         ;Save current mask
                STA     0,PPSAV+6
                LDA     0,CN7           ;Set mask bits 13,14,15 (punch, teletype in and out)
                STA     0,CMASK         ;Set new current mask
                DOBS    0,CPU           ;MSKO and turn on interrupt

                .                       ;Service punch
                .
                .

                INTDS                   ;Turn off interrupt
                LDA     0,PPSAV+6       ;Restore previous mask
                STA     0,CMASK
                MSKO    0
                LDA     0,PPSAV+4       ;Restore Carry
                MOVR    0,0
                LDA     0,PPSAV         ;Restore ACs
                LDA     1,PPSAV+1
                LDA     2,PPSAV+2
                LDA     3,PPSAV+3
                INTEN                   ;Turn on interrupt
                JMP     @PPSAV+5        ;Restore PC
```

where the reader service routine returns to TEST+2 and all others return to TEST. The fastest device, the reader, will never be delayed too much. But suppose the program has a significant amount of computing to do. Then we must use the interrupt, but what about the priority structure? If input-output service for the teletype (as in the sample master routine above) requires 1 ms and punch service requires .8 ms, then reader service will never be delayed more than 1 ms if we simply turn the interrupt off while servicing each device. But if teletype service requires 30 ms per character, then neither reader nor punch will be able to run at full speed unless we use the priority structure as illustrated in the sample routine.

Programming Suggestions. A convenient method for handling a large number of priority levels is to use a pushdown list for saving the machine state. This obviates setting aside so many specific locations for saving accumulators and the like, and makes it very easy for a routine at any level in a sequence of nested routines to restore the state for the interrupted program. If many devices are in use it may frequently happen that when one routine is dismissing an interrupt, a device of lower priority is already waiting. Thus much time might be wasted in restoring the machine state only to have to save it again as soon as the interrupt is turned back on. The devices of concern in this situation are those with priority less than or equal to the device presently being serviced, but of priority greater than that of the device whose routine is about to be resumed (to which the current dismissal will return). The usual dismissal procedure (as illustrated in the sample master routine given above) begins by disabling the interrupt and restoring the previous mask. If the program then gives an

```
INTA    AC
```

a device code will be read into AC if any device of priority higher than that of the interrupted routine has requested service. Since this means that the device will interrupt before the interrupted program can restart, the current program can save a great deal of time by servicing the higher priority device without bothering to restore and resave the machine state. If AC is clear after the INTA is given, no device of appropriate priority has requested service, and the current routine can proceed with the usual dismissal.

Remember the following when programming an interrupt routine:

- An interrupt cannot be started until the current instruction is finished. Therefore do not use lengthy indirect address chains if devices that require very fast service can request an interrupt.
- The routine must save the accumulators and the Carry flag if these will be used by it.
- If this interrupt routine can itself be interrupted, then it must save location 0 so PC can later be restored properly.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. *Eg* computations that can be performed outside the routine should not be included within it.
- The routine should restore the accumulators and Carry when returning to the interrupted program.

2.5 DATA CHANNEL

The maximum rate for data transfers between external devices and core memory could be no greater than 80,000 words per second if the transfers were executed under program control. To allow rates greater than 200,000 the processor contains a data channel through which data can be transferred automatically using only one processor cycle per word. At lower rates the channel also frees processor time to allow execution of a program concurrently with data transfers for a device.

Besides the straightforward transfer of a word between memory and a device in either direction, the data channel also allows a device to increment by one a word already in memory and to add a word to the contents of a memory location. In these two cases involving an arithmetic operation, the processor sends an overflow

signal to the device if the operation should increase the contents of the memory location above $2^{16} - 1$. The data channel is used by devices requiring very high data transfer rates, such as magnetic tape or disk, and by devices requiring the specialized transfer functions. *Eg* the memory increment feature would be used for pulse height analysis, the add-to-memory feature for signal averaging.

The program cannot affect the data channel directly because there are no instructions for it; instead the program sets up the device to use the channel. When the device requires data service, it requests access to memory via the data channel. At the beginning of every memory cycle the processor synchronizes any requests that are then being made. As soon as the processor completes an instruction it takes care of all requests that have been synchronized or are synchronized while it is handling transfers. If several devices are waiting for service simultaneously, the first to receive it is the one that is physically closest to the processor on the bus. After taking care of all data channel requests, the processor starts an interrupt if a device is waiting for one; otherwise it resumes the execution of instructions.

Timing. The time a device must wait for data channel access depends on when its request is made within an instruction and how many devices of higher priority are also requesting access. Once the processor finishes an instruction, a given device must wait until all devices closer than it on the bus have been serviced: the highest priority device can preempt all processor time if it requests access at the maximum rate. At less than the maximum rate the closest device need wait no longer than the time required for the processor to finish the instruction that is being performed when the request is made. The maximum instruction time for a 4K memory is about 6 μs with direct addressing, but an additional 2.6 μs is required for every level of indirect addressing.

CAUTION

Devices that use the data channel often require service very quickly. Since a device must always wait for the current instruction to end, do not use lengthy indirect addressing chains when the data channel is in use.

The times required for the various data channel transfer functions are as follows.

Function	*Time in microseconds*		*Transfers per second*
	4K memory	*General case*	*with 4K memory*
Data in	3.5	$M + .9$	285,500
Data out	4.4	$M + 1.8$	227,500
Increment memory	4.4	$M + 1.8$	227,500
Add to memory	5.3	$M + 2.7$	187,500

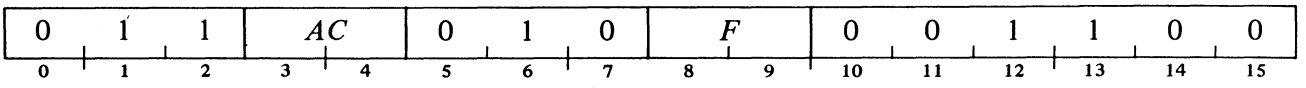
2.6 PROCESSOR OPTIONS

Optional equipment for the processor includes a real time clock and a power monitor with facility for automatic restart after power failure.

Real Time Clock

The clock generates a sequence of pulses that is independent of processor timing. It uses only one IO transfer instruction to set the clock frequency. Busy and Done are controlled or sensed by bits 8 and 9 in all IO instructions with device code 14, mnemonic RTC. Interrupt Disable is controlled by interrupt priority mask bit 13.

DOA –,RTC **Data Out A, Real Time Clock** 4.7 μs



Perform the function specified by *F* and select the clock frequency by AC bits 14 and 15 as follows.

AC bits 14–15	*Frequency*
00	Ac line frequency
01	10 Hz
10	100 Hz
11	1000 Hz

Time. $M + 2.1$ μs

Setting Busy allows the next pulse from the clock to set Done, requesting an interrupt if Interrupt Disable is clear. A DOA to select the frequency need by given only once; following each interrupt an NIOS sets up the clock for the next pulse.

When Busy is first set the first interrupt can come at any time up to the clock period. But once one interrupt has occurred, further interrupts are at the clock frequency provided that the program always sets Busy before the next period expires.

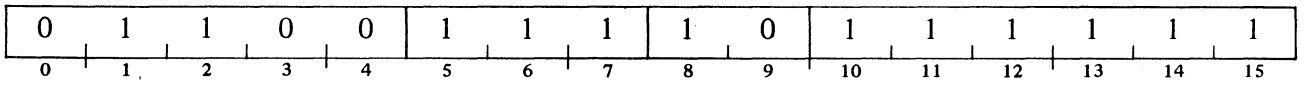
The clock is used primarily for low resolution timing (compared to processor speed) but it has high long-term accuracy. Power turnon and the IO reset function generated by the program or from the console reset the clock to line frequency.

Power Monitor and Autorestart

When ac power is turned on, memory is unaltered, the initial states of PC, the accumulators and flags are indeterminate, and the computer is stopped. If ac power should fail there is a delay of 1 to 2 milliseconds before the processor shuts down. In so doing, the processor always completes a memory cycle and sequences power off so the contents of memory are unaffected. The optional power monitor warns the program when power is failing by setting the Power Failure flag. This action automatically requests an interrupt — there is no interrupt disable flag for the power monitor. Of course the interrupt must be on if a power failure is to produce an interrupt.

The power monitor does not respond to the INTA instruction. Thus when an interrupt occurs in a machine equipped with the power monitor, the program should test the Power Failure flag before giving INTA or testing other devices. The flag corresponds to the Done flag and is tested by either of these instructions.

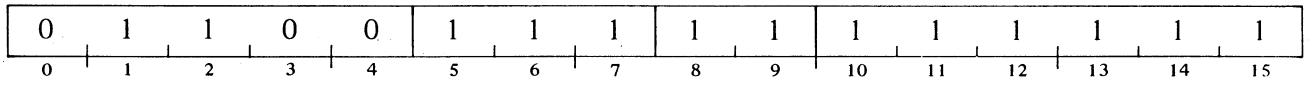
SKPDN CPU **Skip if Power Failure is Nonzero** 4.4 μs



Skip the next instruction in sequence if Power Failure is 1.

Time. $M + 1.8$ μs

SKPDZ CPU **Skip if Power Failure is Zero** 4.4 μs



Skip the next instruction in sequence if Power Failure is 0.

Time. $M + 1.8$ μs

If the power does fail the program should save the accumulators and Carry in memory, save location 0 (for restoring PC in the interrupted program), put a JMP to the desired restart location in location 0, and then HALT.

The action taken by the processor when an adequate power level is restored depends on the power switch on the operator console. If the switch is on, power comes back on with the machine stopped. If the switch is in the lock position, then 200 ms after power comes back on the processor executes a JMP 0, which causes it to begin executing instructions in normal sequence at location 0.

2.7 OPERATION

The operator console is illustrated on page 1-2. The lights in the upper right display control conditions, the rows of lights in the upper center display the processor registers. Below the latter is a register of toggle switches through which the operator can supply addresses and data to the processor (the up position of a switch represents a 1). The register can be used in conjunction with some of the operating switches, and its contents are read by the READS instruction.

In the row at the bottom of the panel are the operating switches. Each switch lever is actually two momentary-contact logical switches with a common off position in the center. Lifting the lever up turns on the switch whose name is printed above it; pressing it down turns on the switch whose name is written below.

At the upper left is a 3-position key-operated rotary switch that controls power and locks the console. Turning it to ON simply turns on power. Turning it to LOCK keeps power on and disables the operating switches so no one can interfere with the operation of the processor (the operator can still use the data switches to supply information to the program).

Indicators. When any indicator is lit the associated flipflop is in the 1 state or the associated function is true. A few indicators display useful information while the processor is running, but most change too frequently and are therefore discussed in terms of the information they display when the processor has stopped.

The instruction lights display the left eight bits of the instruction being executed or just completed (these lights are all off if the processor stops following a program interrupt or data channel cycle). The address lights display the contents of PC, and the data lights display the contents of the Carry flag and the memory buffer.

RUN	The processor is in normal operation with one instruction following another. When the light goes off, the computer stops.
ION	The program interrupt is enabled (this is the Interrupt On flag).
FETCH	The next processor cycle will be used to fetch an instruction from memory.
DEFER	The next processor cycle will be used to fetch an address word in an indirectly addressed memory reference instruction.
EXECUTE	The next processor cycle will be used to reference memory for an operand in a move data or modify memory instruction.

DCH — The next processor cycle will be used by the data channel for direct access to memory by an in-out device.

PI — The next processor cycle will be used to start an interrupt by storing PC in location 0.

Operating Switches. All of the switches in the bottom row except STOP and RESET are interlocked so that they have no effect if RUN is lit. The four pairs of switches at the left are for depositing data in the accumulators and examining their contents. Lifting a switch lever up loads the contents of the data switches into the specified accumulator; pressing it down displays the contents of the accumulator in the data lights. At completion FETCH is lit and the instruction lights are off.

The ten switches at the right perform the following functions when turned on.

EXAMINE — Load the address contained in the data switches into PC (which is displayed in the address lights) and display the contents of the addressed location in the data lights. At completion FETCH is lit.

DEPOSIT — Deposit the contents of the data switches in the memory location specified by the address lights. At completion FETCH is lit and the data lights display the word deposited.

EXAMINE NEXT — Add 1 to the PC address displayed in the address lights and display the contents of the location specified by the incremented address in the data lights. At completion FETCH is lit.

DEPOSIT NEXT — Add 1 to the PC address displayed in the address lights and deposit the contents of the data switches in the memory location specified by the incremented address. At completion FETCH is lit and the data lights display the word deposited.

The above four switches can be used for a sequence of operations on consecutive memory locations. The sequence must begin with EXAMINE to supply the initial address unless PC already points to the right location. Suppose we set the data switches to octal 100 initially. Then the following sequence of switch settings produces the effects listed.

EXAMINE	Display location 100.
EXAMINE NEXT	Display location 101.
EXAMINE NEXT	Display location 102.
DEPOSIT	Load data switches into 102.
EXAMINE NEXT	Display location 103.
DEPOSIT	Load data switches into 103.
DEPOSIT NEXT	Load data switches into 104.
EXAMINE NEXT	Display location 105.

START — Load the address contained in the data switches into PC, light FETCH and RUN, and begin normal operation by executing the instruction at the location specified by PC.

STOP — Stop with FETCH on before beginning the next instruction. Thus the processor finishes the current instruction, and then stops with the instruction lights displaying the instruction, unless a device is waiting for data channel access or a program interrupt, in which case it performs all such operations before stopping with the instruction lights off. The address lights point to the next instruction.

CAUTION

If the current instruction contains an infinitely long indirect addressing chain or there are continuous data channel requests, pressing STOP will *not* stop the computer (see RESET, below).

CONTINUE — Turn on RUN and begin normal operation in the state indicated by the lights.

INST STEP — Begin operation in the state indicated by the lights but then stop as though STOP had been pressed at the same time. If the stop occurs at the end of an instruction, the data displayed by the data lights depends on the instruction as follows.

LDA, STA, ISZ, DSZ	Operand
JMP	Effective address
JSR	The address loaded into AC3 (old PC + 1).
Arithmetic and logical	Instruction
In-out	Instruction

Note that the AC switches can be used between instruction steps without requiring any readjustment.

MEMORY STEP — Perform a single processor cycle in the state indicated by the lights and then stop. At completion the lights will point to the next state to be executed.

CAUTION

Using the AC switches between memory steps within an instruction usually destroys information necessary for the execution of the rest of the instruction.

RESET — Stop at the end of the current processor cycle. Clear the flags in all IO devices, clear Interrupt On, and set the clock to line frequency.

EXAMINE can be used to load PC for beginning any single step procedure. Instruction stepping can also be begun by pressing START *while* holding STOP on.

To use the various examine and deposit switches between instruction steps, simply remember what PC is and restore it before continuing.

Chapter III
Hardcopy Equipment

This chapter discusses the simpler peripheral devices: teletypewriter, tape reader, tape punch, card reader, card punch, plotter and line printer. These devices are used primarily for communication between computer and operator using a paper medium: tape, cards, form paper or graph paper. All transfers for them are made by the program through the accumulators.

The program can type out characters on the teletypewriter and can read characters that have been typed in at the keyboard. This device has the slowest transfer rate of any, but it provides a convenient means of man-machine interaction. The KSR teletypewriters comprise only a keyboard and printer; the ASR models also have a slow tape reader and punch. This punch and the separate high speed punch supply output in the form of 8-channel perforated paper tape. The information punched in the tape can be brought into the processor by the high speed tape reader or the one mounted in the teletypewriter.

The card equipment processes standard 12-row 80-column cards. Many programmers find cards a convenient medium for source program input and for supplying data that varies from one program run to another. Cards and paper tape are both convenient to prepare manually, but card input is much faster than tape, and simple changes are easier to make: individual cards can be repunched, and cards can be added or removed from the deck. A possible consideration in using cards is that many installations do not include an on line card punch.

The line printer provides text output at a relatively high rate. The program must effectively typeset each line; upon command the printer then prints the entire line. With the plotter, the program can produce ink drawings by controlling the incremental motion of pen on paper in a cartesian coordinate system. Curves and figures of any shape can be generated by proper combinations of motion in x and y.

3.1 TELETYPEWRITER

Four teletypewriter models are regularly available for use with the NOVA: the ASR33, KSR33 and KSR35, all of which are capable of speeds up to ten characters per second, and the KSR37, which can handle up to fifteen characters per second. The program can type out characters and can read in the characters produced when keys are struck at the keyboard. With an ASR the program can also punch characters in a tape and read characters from a tape.

The teletype separates its input and output functions and is really two distinct devices. Each has its own device code, its own Busy, Done and Interrupt Disable flags, and its own interrupt priority mask assignment. Placing a code for a character in the output buffer and setting Output Busy causes the teletype to print the character or perform the designated control function. Striking a key places the code for the associated character in the input buffer where it can be retrieved by the program, but it does nothing at the teletype unless the program sends the code back as output.

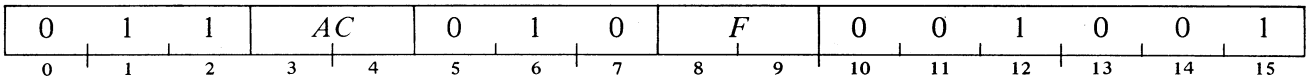
Character codes received from the keyboard have eight bits wherein the most significant is an even parity bit. The Model 33 and 35 printers ignore the parity bit in characters transmitted to them. The model 37 ignores the parity bit in a code for a printable character, but it performs no function when it receives a control code with incorrect parity.

The Model 37 has the entire character set listed in the table in Appendix E. Lower case characters are not available on the Model 33 or 35, but transmitting a lower case code to the teletype causes it to print the corresponding upper case character. (There are, of course, no restrictions on the codes that can be punched in or read from tape.) To go to the beginning of a new line the program must send both a carriage return, which moves the type block or box to the left margin, and a line feed, which spaces the paper. The horizontal and vertical tabs and form feed have no effect on the Model 33 printer.

Teletype Output

The teletype output uses only one IO transfer instruction. Output Busy and Output Done are controlled or sensed by bits 8 and 9 in all IO instructions with device code 11, mnemonic TTO. Output Interrupt Disable is controlled by interrupt priority mask bit 15.

DOA –,TTO **Data Out A, Teletype Output** 4.7 μs



Load the contents of AC bits 8–15 into the teletype output buffer, and perform the function specified by *F*.

Time. $M + 2.1$ μs

Setting Output Busy turns on the transmitter, causing it to send the contents of the output buffer serially to the teletype (the buffer is cleared during transmission). The printer prints the character or performs the indicated control function. If the punch is on, the character is also punched in the tape, with AC bit 15 corresponding to channel 1 (a 1 in AC produces a hole in the tape). Completion of transmission clears Output Busy and sets Output Done, requesting an interrupt if Output Interrupt Disable is clear.

NOTE

Although the buffer clears during transmission, giving an NIOS without loading it again does not transmit a zero character. So do not give an NIOS without first loading the buffer. To transmit any character including null, either give a DOAS or give a DOA followed by an NIOS.

CAUTION

Clearing Output Busy while the transmitter is running (as with an NIOC) terminates the transmission. But the printer still prints whatever character is represented by the indeterminate code it receives.

Timing. Models 33 and 35 can type or punch up to ten characters per second. After Output Done is set, the program has 4.55 ms to give a DOAS to keep typing or punching at the maximum rate. The 37 can handle fifteen characters per second, 66.7 ms per character. After Output Done is set, the program has 3.33 ms to send a new character to maintain the maximum typing rate.

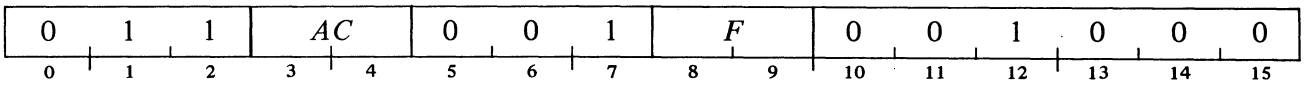
The sequence carriage return-line feed, when given in that order, allows sufficient time for the type block

to get to the beginning of a new line. After tabbing, the program must wait for completion of the mechanical function by sending one or two rubouts. If the time is critical, the programmer should measure the time required for his tabs. Tabs are normally set every eight spaces (columns 9, 17, . . .) and require one rubout.

Teletype Input

The teletype input uses only one IO transfer instruction. Input Done is controlled or sensed by bits 8 and 9 in all IO instructions with device code 10, mnemonic TTI, but there is no Input Busy. Nevertheless the start function is used: programming 01 in bits 8 and 9 of an NIO or a transfer instruction places the reader in operation. Input Interrupt Disable is controlled by interrupt priority mask bit 14. SKPBZ TTI is an absolute skip, SKPBN TTI is a no-op.

DIA –,TTI **Data In A, Teletype Input** 4.4 μs



Transfer the contents of the input buffer into AC bits 8–15, and perform the function specified by *F*.

Time. $M + 1.8$ μs

Reception from the keyboard requires no initiating action by the program; striking a key transmits the code for the character serially to the input buffer. However, if the reader is under program control, giving the start function (NIOS or DIAS) causes the reader to read all eight channels from the next line on tape and transmit the line serially into the buffer (the presence of a hole produces a 1 in the buffer). Completion of reception sets Input Done, requesting an interrupt if Output Interrupt Disable is clear. When the character is brought into AC, tape channel 1 corresponds to AC bit 15.

Timing. After Input Done is set by a Model 33 or 35, the character is available for retrieval by a DIA for 21.59 ms before another key strike can destroy it. If the reader is in use, the program has 3.41 ms to give a DIAS (or DIA and NIOS) and keep the tape in continuous motion. With the 37, the character is available for 9.17 ms after Input Done is set.

Programming Examples

There are basically two procedures for using the skip instructions in a loop to process a series of characters. Consider this loop for typing out (we assume the printer is not in use).

```
OUT:    DOAS    AC,TTO      ;Type out
        SKPDN   TTO         ;Wait till transmission done
        JMP     .-1
        .                   ;Get next character, compute, etc
        .
        .
        JMP     OUT         ;Go back
```

This procedure is very poor as most of the time is spent waiting during the transmission, and there is very little time to do anything afterwards if we are to go back to type out the next character at full speed. But with this arrangement:

```
OUT:    SKPBZ   TTO         ;Wait till printer free
        JMP     .-1
        DOAS    AC,TTO      ;Type out character
        .                   ;Compute, etc
        .
        .                   ;Get next character
        JMP     OUT         ;Go back
```

we have almost all of the time for worthwhile program and we can run at full speed provided only that we jump back to OUT before the entire teletype cycle time is over. Also, the first time into the loop we wait until any previous (perhaps unknown to us) teletype output operation is finished.

The same dichotomy exists for input operations. This is bad:

```
IN:     NIOS    TTI         ;Read character
        SKPDN   TTI         ;Wait till reception done
        JMP     .-1
        DIA     AC,TTI      ;Bring in character
        .                   ;Decide whether to read another character, etc
        .
        .
        JMP     IN          ;Go back
```

but this is good:

```
        NIOS    TTI         ;Read first character
IN:     SKPDN   TTI         ;Wait till reception done
        JMP     .-1
        DIAS AC,TTI         ;Bring in character and read another
        .                   ;Compute, etc
        .
        .
        JMP     IN          ;Go back
```

Of course the last program does not allow us to inspect a character to determine whether to get another one. So for the best of all possible worlds we combine the procedures.

```
IN:     NIOS    TTI         ;Read character
        .                   ;Lots of time to compute
        .
        .
        SKPDN   TTI         ;Wait till reception done
        JMP     .-1
        DIA     AC,TTI      ;Bring in character
        .                   ;Decide whether to get another
        .
        .
        JMP     IN          ;Do this if want another
        .                   ;Skip to here if not
        .
        .
```

Operation

A KSR is actually two independent devices, keyboard and printer, which can be operated simultaneously. An ASR is really four devices, keyboard, printer, reader and punch, which can be operated in various combi-

nations. Power must be turned on by the operator. On the 33 and 35 the switch is beside the keyboard and is labeled LINE/OFF/LOCAL or ON/OFF and has an unmarked third position opposite ON. A similar switch is located beneath the stand on the 37. When this switch is set to LOCAL or the unmarked position, power is on but the machine is off line and can be used like a typewriter. Moreover, in an ASR, turning on the punch allows the operator to punch a tape from the keyboard, and running the reader allows a tape to control the printer (if the punch is also on, it duplicates the tape).

Turning the switch to LINE or ON connects the unit to the computer and separates its input and output functions. Thus any information transmitted to the computer from the keyboard affects the printer only insofar as the computer sends it back. Turning on the reader places it under program control, and turning on the punch causes it to punch whatever is sent to the printer by the computer.

The only control on the reader is a 3-position switch. When the switch is in the FREE position, the tape can be moved by hand freely through the reader mechanism. The STOP position engages the reader clutch so the tape is stationary but the reader is still off. Turning the switch to START causes the reader to read the tape if the unit is in local, but places it under program control if on line.

The operator controls the punch by means of four pushbuttons. The two on the right turn the punch on and off. Pressing the REL. button releases the tape so it can be moved by hand through the punch mechanism. Pressing B. SP. moves the tape backward one frame so the operator can delete a frame that is incorrect by striking the rubout key. Pressing HERE IS with the keyboard in local punches twenty lines of blank tape (lines with only a feed hole punched).

The keyboard resembles that of a standard typewriter. Codes for printable characters on the upper parts of the key tops on the 33 and 35 are transmitted by using the shift key; most control codes require use of the control key. Those familiar with the 33 or 35 who are using the 37 for the first time should take a close look at the keyboard. On the 37 the shift is used for real upper case characters. The control key is used for some control characters, but many have separate keys. Note also that both the keyboard arrangement and the labels differ somewhat. On all models the line feed (labeled "new line" on the 37) spaces the paper vertically at six lines to the inch, and must be combined with a return to start a new line. The local advance (feed) and return keys affect the printer directly and do not transmit codes. Appendix E lists the complete teletype code, ASCII characters, key combinations, and differences among the several models.

On the 33 and 35 is a repeat button REPT. Pressing this button and striking any character key causes transmission of the corresponding code so long as REPT is held down. Characters that require the shift key may also be repeated in this manner, but there is no repetition of control characters.

Teletype manuals supplied with the equipment give complete, illustrated descriptions of the procedures for loading paper and tape, changing the ribbon, and setting horizontal and vertical tabs. Setting tabs is usually left for maintenance personnel; in any event, the best and easiest way to learn how to do any of these things is to have someone who knows show you how. However, as a precautionary measure we describe here the things you may have to do yourself.

Tape. The tape moves in the reader from back to front with the feed holes closer to the left edge. To load tape, set the switch to FREE, release the cover guard by opening the latch at the right, place the tape so that the sprocket wheel teeth engage the feed holes, close the cover guard, and set the switch to STOP.

To load tape in the punch, raise the cover, feed the tape manually from the top of the roll into the guide at the back, move the tape through the punch by turning the friction wheel, then close the cover. Turn on the punch with the unit in local and punch about two feet of leader by pressing HERE IS or the control, shift and P keys to generate null codes.

Paper. The 33 printer has an 8½-inch roll of paper at the back. Printed sections can be torn off against the edge of the glass window in front of the platen. To replenish the paper, snap open the cover, remove the old roll and slip a new one in its place. Draw the paper from the roll around the platen as in an ordinary typewriter.

The 35 and 37 printers have a sprocket feed and use 8½ X 11 fanfold form paper. The supply is held in a tray at the back. To replenish it, first remove the upper cover by pressing the cover release button on the right side. To free the remaining old paper for removal, lift the paper guides by pushing the handle marked PUSH at the right of the platen. To insert new paper from the tray, bring it up below the platen at the rear, line up the holes at the edges of the paper with the sprockets, and press line feed (in local) to draw the paper under the platen.

Ribbon. Replace the ribbon whenever it becomes worn or frayed or the printing becomes too light. Disengage the old ribbon from the ribbon guides on either side of the type block, and remove the reels by lifting the spring clips on the reel spindles and pulling the reels off. Remove the old ribbon from one of the reels and replace the empty reel on one side of the machine; install a new reel on the other side. Push down both reel spindle spring clips to secure the reels. Unwind the fresh ribbon from the inside of the supply reel, over the guide roller, through the two guides on either side of the type block, out around the other guide roller, and back onto the inside of the takeup reel. Engage the hook on the end of the ribbon over the point of the arrow in the hub. Wind a few turns of the ribbon to make sure that the reversing eyelet has been wound onto the spool. Make sure the ribbon is seated properly and feeds correctly in operation.

Appendices

THE FOLLOWING APPENDICES WILL BE ADDED

A Interfacing

I. IO Bus

II. Interface Timing

III. Design of Interface Equipment

B Installation

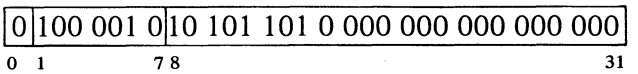
APPENDIX C

FLOATING POINT ARITHMETIC

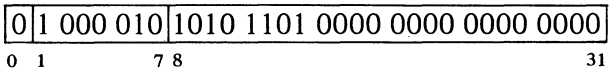
Software is available for processing floating point numbers. For a given word length, floating point format sacrifices some precision for a much greater range in order of magnitude. The software interprets the two-word floating point representation of a number as containing a sign (bit 0), a 7-bit characteristic, and a 24-bit proper fraction. The characteristic is the coded exponent of the power of 16 that the fraction must be multiplied by to give the number being represented.

For a positive number the sign is 0. The contents of bits 8–31 are interpreted as a binary fraction (it may often be convenient to view this as six 4-bit hexadecimal digits), and the contents of bits 1–7 are interpreted as an integral exponent in excess 64 (100_8) code. Exponents from -64 to $+63$ are therefore represented by the binary equivalents of 0–127 (0–177_8). The negative of a number is obtained simply by changing the sign bit to 1 — the rest of the number remains in positive form. Zero is represented by all 0s in sign, characteristic and fraction. The routines always represent a zero result in this form (referred to as "true" zero), but they interpret any operand with a zero fractional part as being zero.

$+173_{10} = +255_8 =$

$+.532_8 \times 16^2 =$ 

=

$+.10\,13_{16} \times 16^2 =$ 

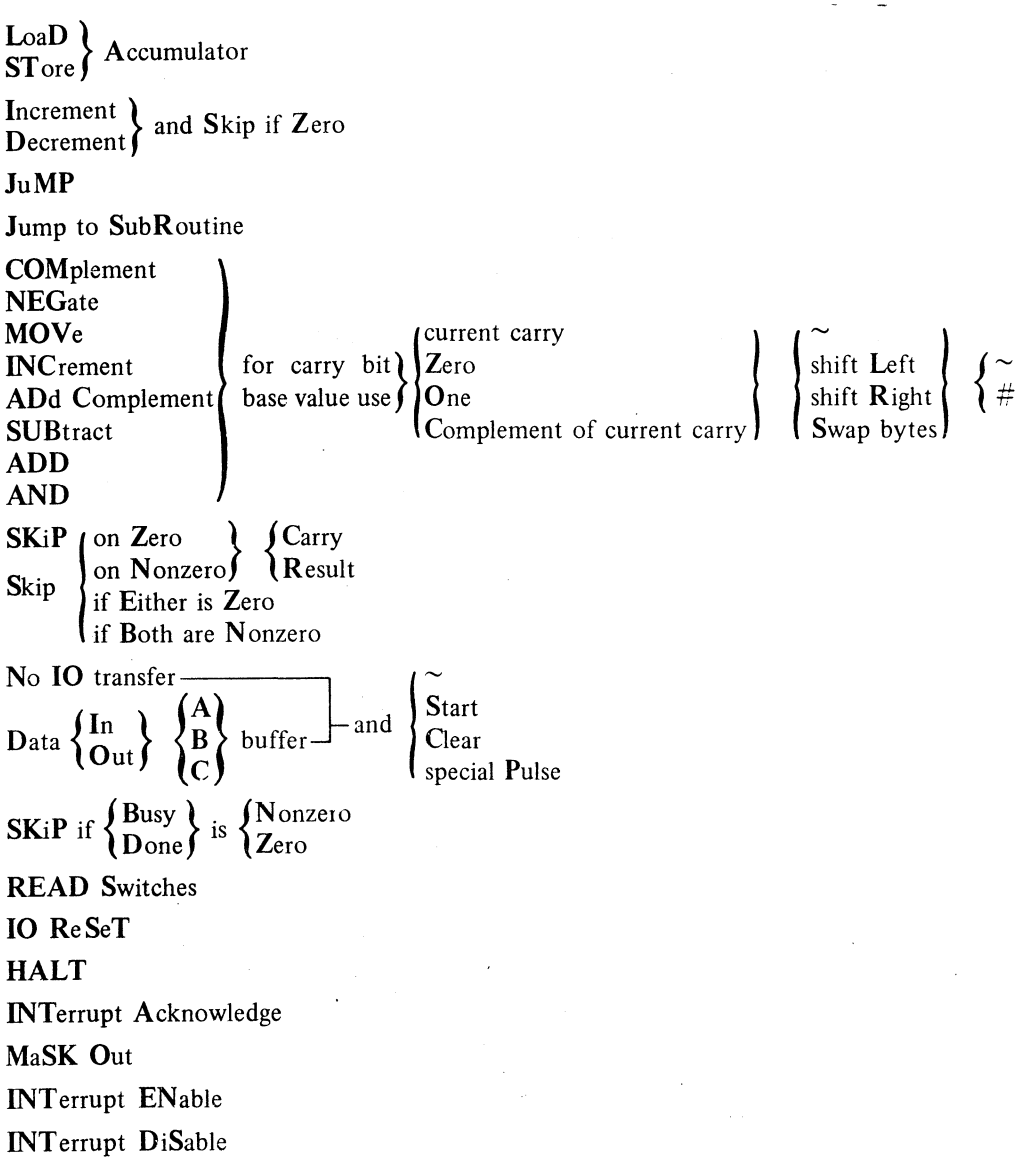
Most routines assume that all nonzero operands are normalized, and they normalize a nonzero result. A floating point number is considered normalized if the fraction is greater than or equal to 1/16 and less than 1; in other words it has a 1 in the first four bits (bits 8–11 of the high order word). These numbers thus have a fractional range of 1/16 to $1-2^{-24}$ ($1-16^{-6}$) and an exponent range of -64 to $+63$. This corresponds to a decimal range of approximately 2.4×10^{-78} to 7.2×10^{75}.

APPENDIX D

INSTRUCTION MNEMONICS

The table beginning on the next page lists the instruction mnemonics in numerical order. Following that is a listing in alphabetical order that gives the octal value, a short description of the instruction, the number of the page on which the full description appears in Chapter 2, and the instruction time in microseconds with a 4K memory.

The derivation of the instruction mnemonics is as follows.



INSTRUCTION MNEMONICS

NUMERIC LISTING

000000	JMP	062677	IORST	100370	COMCS#
000001	SKP	062700	DICP	100400	NEG
000002	SZC	063000	DOC	100410	NEG#
000003	SNC	063077	HALT	100420	NEGZ
000004	SZR	063100	DOCS	100430	NEGZ#
000005	SNR	063200	DOCC	100440	NEGO
000006	SEZ	063300	DOCP	100450	NEGO#
000007	SBN	063400	SKPBN	100460	NEGC
000010	#	063500	SKPBZ	100470	NEGC#
002000	@	063600	SKPDN	100500	NEGL
004000	JSR	063700	SKPDZ	100510	NEGL#
010000	ISZ	100000	@	100520	NEGZL
014000	DSZ	100000	COM	100530	NEGZL#
020000	LDA	100010	COM#	100540	NEGOL
040000	STA	100020	COMZ	100550	NEGOL#
060000	NIO	100030	COMZ#	100560	NEGCL
060100	NIOS	100040	COMO	100570	NEGCL#
060177	INTEN	100050	COMO#	100600	NEGR
060200	NIOC	100060	COMC	100610	NEGR#
060277	INTDS	100070	COMC#	100620	NEGZR
060300	NIOP	100100	COML	100630	NEGZR#
060400	DIA	100110	COML#	100640	NEGOR
060477	READS	100120	COMZL	100650	NEGOR#
060500	DIAS	100130	COMZL#	100660	NEGCR
060600	DIAC	100140	COMOL	100670	NEGCR#
060700	DIAP	100150	COMOL#	100700	NEGS
061000	DOA	100160	COMCL	100710	NEGS#
061100	DOAS	100170	COMCL#	100720	NEGZS
061200	DOAC	100200	COMR	100730	NEGZS#
061300	DOAP	100210	COMR#	100740	NEGOS
061400	DIB	100220	COMZR	100750	NEGOS#
061477	INTA	100230	COMZR#	100760	NEGCS
061500	DIBS	100240	COMOR	100770	NEGCS#
061600	DIBC	100250	COMOR#	101000	MOV
061700	DIBP	100260	COMCR	101010	MOV#
062000	DOB	100270	COMCR#	101020	MOVZ
062077	MSKO	100300	COMS	101030	MOVZ#
062100	DOBS	100310	COMS#	101040	MOVO
062200	DOBC	100320	COMZS	101050	MOVO#
062300	DOBP	100330	COMZS#	101060	MOVC
062400	DIC	100340	COMOS	101070	MOVC#
062500	DICS	100350	COMOS#	101100	MOVL
062600	DICC	100360	COMCS	101110	MOVL#

101120	MOVZL	101700	INCS	102460	SUBC
101130	MOVZL#	101710	INCS#	102470	SUBC#
101140	MOVOL	101720	INCZS	102500	SUBL
101150	MOVOL#	101730	INCZS#	102510	SUBL#
101160	MOVCL	101740	INCOS	102520	SUBZL
101170	MOVCL#	101750	INCOS#	102530	SUBZL#
101200	MOVR	101760	INCCS	102540	SUBOL
101210	MOVR#	101770	INCCS#	102550	SUBOL#
101220	MOVZR	102000	ADC	102560	SUBCL
101230	MOVZR#	102010	ADC#	102570	SUBCL#
101240	MOVOR	102020	ADCZ	102600	SUBR
101250	MOVOR#	102030	ADCZ#	102610	SUBR#
101260	MOVCR	102040	ADCO	102620	SUBZR
101270	MOVCR#	102050	ADCO#	102630	SUBZR#
101300	MOVS	102060	ADCC	102640	SUBOR
101310	MOVS#	102070	ADCC#	102650	SUBOR#
101320	MOVZS	102100	ADCL	102660	SUBCR
101330	MOVZS#	102110	ADCL#	102670	SUBCR#
101340	MOVOS	102120	ADCZL	102700	SUBS
101350	MOVOS#	102130	ADCZL#	102710	SUBS#
101360	MOVCS	102140	ADCOL	102720	SUBZS
101370	MOVCS#	102150	ADCOL#	102730	SUBZS#
101400	INC	102160	ADCCL	102740	SUBOS
101410	INC#	102170	ADCCL#	102750	SUBOS#
101420	INCZ	102200	ADCR	102760	SUBCS
101430	INCZ#	102210	ADCR#	102770	SUBCS#
101440	INCO	102220	ADCZR	103000	ADD
101450	INCO#	102230	ADCZR#	103010	ADD#
101460	INCC	102240	ADCOR	103020	ADDZ
101470	INCC#	102250	ADCOR#	103030	ADDZ#
101500	INCL	102260	ADCCR	103040	ADDO
101510	INCL#	102270	ADCCR#	103050	ADDO#
101520	INCZL	102300	ADCS	103060	ADDC
101530	INCZL#	102310	ADCS#	103070	ADDC#
101540	INCOL	102320	ADCZS	103100	ADDL
101550	INCOL#	102330	ADCZS#	103110	ADDL#
101560	INCCL	102340	ADCOS	103120	ADDZL
101570	INCCL#	102350	ADCOS#	103130	ADDZL#
101600	INCR	102360	ADCCS	103140	ADDOL
101610	INCR#	102370	ADCCS#	103150	ADDOL#
101620	INCZR	102400	SUB	103160	ADDCL
101630	INCZR#	102410	SUB#	103170	ADDCL#
101640	INCOR	102420	SUBZ	103200	ADDR
101650	INCOR#	102430	SUBZ#	103210	ADDR#
101660	INCCR	102440	SUBO	103220	ADDZR
101670	INCCR#	102450	SUBO#	103230	ADDZR#

103240	ADDOR	103430	ANDZ#	103620	ANDZR
103250	ADDOR#	103440	ANDO	103630	ANDZR#
103260	ADDCR	103450	ANDO#	103640	ANDOR
103270	ADDCR#	103460	ANDC	103650	ANDOR#
103300	ADDS	103470	ANDC#	103660	ANDCR
103310	ADDS#	103500	ANDL	103670	ANDCR#
103320	ADDZS	103510	ANDL#	103700	ANDS
103330	ADDZS#	103520	ANDZL	103710	ANDS#
103340	ADDOS	103530	ANDZL#	103720	ANDZS
103350	ADDOS#	103540	ANDOL	103730	ANDZS#
103360	ADDCS	103550	ANDOL#	103740	ANDOS
103370	ADDCS#	103560	ANDCL	103750	ANDOS#
103400	AND	103570	ANDCL#	103760	ANDCS
103410	AND#	103600	ANDR	103770	ANDCS#
103420	ANDZ	103610	ANDR#		

INSTRUCTION MNEMONICS

ALPHABETIC LISTING

			Page	*Time*
ADD	103000	Add ACS to ACD; use Carry as base for carry bit.	2-16	5.9
ADDC	103060	Add ACS to ACD; use complement of Carry as base for carry bit.		5.9
ADDCL	103160	Add ACS to ACD; use complement of Carry as base for carry bit; rotate left.		5.9
ADDCR	103260	Add ACS to ACD; use complement of Carry as base for carry bit; rotate right.		5.9
ADDCS	103360	Add ACS to ACD; use complement of Carry as base for carry bit; swap halves of result.		5.9
ADDL	103100	Add ACS to ACD; use Carry as base for carry bit; rotate left.		5.9
ADDO	103040	Add ACS to ACD; use 1 as base for carry bit.		5.9
ADDOL	103140	Add ACS to ACD; use 1 as base for carry bit; rotate left.		5.9
ADDOR	103240	Add ACS to ACD; use 1 as base for carry bit; rotate right.		5.9
ADDOS	103340	Add ACS to ACD; use 1 as base for carry bit; swap halves of result.		5.9
ADDR	103200	Add ACS to ACD; use Carry as base for carry bit; rotate right.		5.9
ADDS	103300	Add ACS to ACD; use Carry as base for carry bit; swap halves of result.		5.9
ADDZ	103020	Add ACS to ACD; use 0 as base for carry bit.		5.9
ADDZL	103120	Add ACS to ACD; use 0 as base for carry bit; rotate left.		5.9
ADDZR	103220	Add ACS to ACD; use 0 as base for carry bit; rotate right.		5.9
ADDZS	103320	Add ACS to ACD; use 0 as base for carry bit; swap halves of result.		5.9
ADC	102000	Add the complement of ACS to ACD; use Carry as base for carry bit.	2-16	5.9
ADCC	102060	Add the complement of ACS to ACD; use complement of Carry as base for carry bit.		5.9
ADCCL	102160	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; rotate left.		5.9
ADCCR	102260	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; rotate right.		5.9
ADCCS	102360	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; swap halves of result.		5.9
ADCL	102100	Add the complement of ACS to ACD; use Carry as base for carry bit; rotate left.		5.9
ADCO	102040	Add the complement of ACS to ACD; use 1 as base for carry bit.		5.9
ADCOL	102140	Add the complement of ACS to ACD; use 1 as base for carry bit; rotate left.		5.9
ADCOR	102240	Add the complement of ACS to ACD; use 1 as base for carry bit; rotate right.		5.9
ADCOS	102340	Add the complement of ACS to ACD; use 1 as base for carry bit; swap halves of result.		5.9
ADCR	102200	Add the complement of ACS to ACD; use Carry as base for carry bit; rotate right.		5.9

			Page	*Time*
ADCS	102300	Add the complement of ACS to ACD; use Carry as base for carry bit; swap halves of result.	2-16	5.9
ADCZ	102020	Add the complement of ACS to ACD; use 0 as base for carry bit.		5.9
ADCZL	102120	Add the complement of ACS to ACD; use 0 as base for carry bit; rotate left.		5.9
ADCZR	102220	Add the complement of ACS to ACD; use 0 as base for carry bit; rotate right.		5.9
ADCZS	102320	Add the complement of ACS to ACD; use 0 as base for carry bit; swap halves of result.		5.9
AND	103400	And ACS with ACD; use Carry as carry bit.	2-16	5.9
ANDC	103460	And ACS with ACD; use complement of Carry as carry bit.		5.9
ANDCL	103560	And ACS with ACD; use complement of Carry as carry bit; rotate left.		5.9
ANDCR	103660	And ACS with ACD; use complement of Carry as carry bit; rotate right.		5.9
ANDCS	103760	And ACS with ACD; use complement of Carry as carry bit; swap halves of result.		5.9
ANDL	103500	And ACS with ACD; use Carry as carry bit; rotate left.		5.9
ANDO	103440	And ACS with ACD; use 1 as carry bit.		5.9
ANDOL	103540	And ACS with ACD; use 1 as carry bit; rotate left.		5.9
ANDOR	103640	And ACS with ACD; use 1 as carry bit; rotate right.		5.9
ANDOS	103740	And ACS with ACD; use 1 as carry bit; swap halves of result.		5.9
ANDR	103600	And ACS with ACD; use Carry as carry bit; rotate right.		5.9
ANDS	103700	And ACS with ACD; use Carry as carry bit; swap halves of result.		5.9
ANDZ	103420	And ACS with ACD; use 0 as carry bit.		5.9
ANDZL	103520	And ACS with ACD; use 0 as carry bit; rotate left.		5.9
ANDZR	103620	And ACS with ACD; use 0 as carry bit; rotate right.		5.9
ANDZS	103720	And ACS with ACD; use 0 as carry bit; swap halves of result.		5.9
COM	100000	Place the complement of ACS in ACD; use Carry as carry bit.	2-14	5.6
COMC	100060	Place the complement of ACS in ACD; use complement of Carry as carry bit.		5.6
COMCL	100160	Place the complement of ACS in ACD; use complement of Carry as carry bit; rotate left.		5.6
COMCR	100260	Place the complement of ACS in ACD; use complement of Carry as carry bit; rotate right.		5.6
COMCS	100360	Place the complement of ACS in ACD; use complement of Carry as carry bit; swap halves of result.		5.6
COML	100100	Place the complement of ACS in ACD; use Carry as carry bit; rotate left.		5.6
COMO	100040	Place the complement of ACS in ACD; use 1 as carry bit.		5.6
COMOL	100140	Place the complement of ACS in ACD; use 1 as carry bit; rotate left.		5.6
COMOR	100240	Place the complement of ACS in ACD; use 1 as carry bit; rotate right.		5.6

			Page	*Time*
COMOS	100340	Place the complement of ACS in ACD; use 1 as carry bit; swap halves of result.	2-14	5.6
COMR	100200	Place the complement of ACS in ACD; use Carry as carry bit; rotate right.		5.6
COMS	100300	Place the complement of ACS in ACD; use Carry as carry bit; swap halves of result.		5.6
COMZ	100020	Place the complement of ACS in ACD; use 0 as carry bit.		5.6
COMZL	100120	Place the complement of ACS in ACD; use 0 as carry bit; rotate left.		5.6
COMZR	100220	Place the complement of ACS in ACD; use 0 as carry bit; rotate right.		5.6
COMZS	100320	Place the complement of ACS in ACD; use 0 as carry bit; swap halves of result.		5.6
DIA	060400	Data in, A buffer to AC.	2-24	4.4
DIAC	060600	Data in, A buffer to AC; clear device.		4.4
DIAP	060700	Data in, A buffer to AC; send special pulse to device.		4.4
DIAS	060500	Data in, A buffer to AC; start device.		4.4
DIB	061400	Data in, B buffer to AC.	2-24	4.4
DIBC	061600	Data in, B buffer to AC; clear device.		4.4
DIBP	061700	Data in, B buffer to AC; send special pulse to device.		4.4
DIBS	061500	Data in, B buffer to AC; start device.		4.4
DIC	062400	Data in, C buffer to AC.	2-25	4.4
DICC	062600	Data in, C buffer to AC; clear device.		4.4
DICP	062700	Data in, C buffer to AC; send special pulse to device.		4.4
DICS	062500	Data in, C buffer to AC; start device.		4.4
DOA	061000	Data out, AC to A buffer.	2-24	4.7
DOAC	061200	Data out, AC to A buffer; clear device.		4.7
DOAP	061300	Data out, AC to A buffer; send special pulse to device.		4.7
DOAS	061100	Data out, AC to A buffer; start device.		4.7
DOB	062000	Data out, AC to B buffer.	2-24	4.7
DOBC	062200	Data out, AC to B buffer; clear device.		4.7
DOBP	062300	Data out, AC to B buffer; send special pulse to device.		4.7
DOBS	062100	Data out, AC to B buffer; start device.		4.7
DOC	063000	Data out, AC to C buffer.	2-25	4.7
DOCC	063200	Data out, AC to C buffer; clear device.		4.7
DOCP	063300	Data out, AC to C buffer; send special pulse to device.		4.7
DOCS	063100	Data out, AC to C buffer; start device.		4.7
DSZ	014000	Decrement location *E* by 1 and skip if result is zero.	2-6	5.2
HALT	063077	Halt the processor (= DOC 0,CPU).	2-28	4.7
INC	101400	Place ACS + 1 in ACD; use Carry as base for carry bit.	2-15	5.6
INCC	101460	Place ACS + 1 in ACD; use complement of Carry as base for carry bit.		5.6

			Page	*Time*
SKP	000001	Skip (skip function in an arithmetic or logical instruction).	2-13	
SKPBN	063400	Skip if Busy is 1.	2-23	4.4
SKPBZ	063500	Skip if Busy is 0.	2-23	4.4
SKPDN	063600	Skip if Done is 1.	2-23	4.4
SKPDZ	063700	Skip if Done is 0.	2-23	4.4
SNC	000003	Skip if carry bit is 1 (skip function in an arithmetic or logical instruction).	2-13	
SNR	000005	Skip if result is nonzero (skip function in an arithmetic or logical instruction).	2-13	
SZC	000002	Skip if carry is 0 (skip function in an arithmetic or logical instruction).	2-13	
SZR	000004	Skip if result is zero (skip function in an arithmetic or logical instruction).	2-13	
STA	040000	Store AC in location *E*.	2-5	5.5
SUB	102400	Subtract ACS from ACD; use Carry as base for carry bit.	2-16	5.9
SUBC	102460	Subtract ACS from ACD; use complement of Carry as base for carry bit.		5.9
SUBCL	102560	Subtract ACS from ACD; use complement of Carry as base for carry bit; rotate left.		5.9
SUBCR	102660	Subtract ACS from ADC; use complement of Carry as base for carry bit; rotate right.		5.9
SUBCS	102760	Subtract ACS from ACD; use complement of Carry as base for carry bit; swap halves of result.		5.9
SUBL	102500	Subtract ACS from ACD; use Carry as base for carry bit; rotate left.		5.9
SUBO	102440	Subtract ACS from ACD; use 1 as base for carry bit.		5.9
SUBOL	102540	Subtract ACS from ACD; use 1 as base for carry bit; rotate left.		5.9
SUBOR	102640	Subtract ACS from ACD; use 1 as base for carry bit; rotate right.		5.9
SUBOS	102740	Subtract ACS from ACD; use 1 as base for carry bit; swap halves of result.		5.9
SUBR	102600	Subtract ACS from ACD; use Carry as base for carry bit; rotate right.		5.9
SUBS	102700	Subtract ACS from ACD; use Carry as base for carry bit; swap halves of result.		5.9
SUBZ	102420	Subtract ACS from ACS; use 0 as base for carry bit.		5.9
SUBZL	102520	Subtract ACS from ACD; use 0 as base for carry bit; rotate left.		
SUBZR	102620	Subtract ACS from ACD; use 0 as base for carry bit; rotate right.		5.9
SUBZS	102720	Subtract ACS from ACD; use 0 as base for carry bit; swap halves of result.		5.9
@	002000	When this character appears in an instruction, the assembler places a 1 in bit 5 to produce indirect addressing.	2-3	
@	100000	When this character appears with a 15-bit address, the assembler places a 1 in bit 0, making the address indirect.	2-3	
#	000010	Appending this character to the mnemonic for an arithmetic or logical instruction places a 1 in bit 12 to prevent the processor from	2-13	

loading the 17-bit result in Carry and ACD. Thus the result of an instruction can be tested for a skip without affecting Carry or the accumulators.

APPENDIX E

IN-OUT CODES

The table on the next two pages lists the in-out devices, their octal codes, mnemonics and DGC option numbers. The table beginning on page E4 lists the complete teletype code. The lower case character set (codes 140–176) is not available on the Model 33 or 35, but giving one of these codes causes the teletypewriter to print the corresponding upper case character. Other differences between the 33-35 and the 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the computer teletypewriter, and the definitions bear no necessary relation to the use of the codes in conjunction with the NOVA software.

IN-OUT DEVICES

Octal Code	Mnemonic	Priority Mask Bit	Device	Page	Option Number
01					
02					
03					
04					
05					
06					
07					
10	TTI	14	Teletype input	3-1	4010
11	TTO	15	Teletype output		
12	PTR	11	Paper tape reader		4011
13	PTP	13	Paper tape punch		4012
14	RTC	13	Real time clock	2-38	4008
15	PLT	12	Incremental plotter		4017
16	CDR	10	Card reader		4016
17	DIS	13	Display		4015
20					
21					
22					
23					
24					
25					
26					
27					
30					
31					
32					
33					
34					
35					
36					
37					
40					
41					
42					
43					
44					
45					
46					
47					
50					
51					
52					

Octal Code	Mnemonic	Priority Mask Bit	Device	Page	Option Number
53					
54					
55					
56					
57					
60					
61					
62					
63					
64					
65					
66					
67					
70					
71					
72					
73					
74					
75					
76	CPU		Central processor	2-26	4001
77			Power monitor and autorestart	2-39	4006

TELETYPE CODE

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed. Repeats on Model 37. Control shift P on Model 33 and 35.
1	001	SOH	Start of heading; also SOM, start of message. Control A.
1	002	STX	Start of text; also EOA, end of address. Control B.
0	003	ETX	End of text; also EOM, end of message. Control C.
1	004	EOT	End of transmission (END); shuts off TWX machines. Control D.
0	005	ENQ	Enquiry (ENQRY); also WRU, "Who are you?" Triggers identification ("Here is. . .") at remote station if so equipped. Control E.
0	006	ACK	Acknowledge; also RU, "Are you. . .?" Control F.
1	007	BEL	Rings the bell. Control G.
1	010	BS	Backspace; also FEO, format effector. Backspaces some machines. Repeats on Model 37. Control H on Model 33 and 35.
0	011	HT	Horizontal tab. Control I on Model 33 and 35.
0	012	LF	Line feed or line space (NEW LINE); advances paper to next line. Repeats on Model 37. Duplicated by control J on Model 33 and 35.
1	013	VT	Vertical tab (VTAB). Control K on Model 33 and 35.
0	014	FF	Form feed to top of next page (PAGE). Control L.
1	015	CR	Carriage return to beginning of line. Control M on Model 33 and 35.
1	016	SO	Shift out; changes ribbon color to red. Control N.
0	017	SI	Shift in; changes ribbon color to black. Control O.
1	020	DLE	Data link escape. Control P (DC0).
0	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	024	DC4	Device control 4, turns punch or auxiliary off. Control T (AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, error. Control U.
1	026	SYN	Synchronous idle (SYNC). Control V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. Control W.
0	030	CAN	Cancel (CANCL). Control X.
1	031	EM	End of medium. Control Y.
1	032	SUB	Substitute. Control Z.
0	033	ESC	Escape, prefix. This code is also generated by control shift K on Model 33 and 35.
1	034	FS	File separator, Control shift L on Model 33 and 35.
0	035	GS	Group separator. Control shift M on Model 33 and 35.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	036	RS	Record separator. Control shift N on Model 33 and 35.
1	037	US	Unit separator. Control shift O on Model 33 and 35.
1	040	SP	Space.
0	041	!	
0	042	″	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	′	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	052	*	Repeats on Model 37.
0	053	+	
1	054	,	
0	055	–	Repeats on Model 37.
0	056	.	Repeats on Model 37.
1	057	/	
0	060	Ø	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	Repeats on Model 37.
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	Repeats on Model 37.
0	131	Y	
0	132	Z	
1	133	[	Shift K on Model 33 and 35.
0	134	\	Shift L on Model 33 and 35.
1	135	]	Shift M on Model 33 and 35.
1	136	↑	
0	137	←	Repeats on Model 37.
0	140	`	Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	Repeats on Model 37.
1	171	y	
1	172	z	
0	173	{	
1	174	|	
0	175	}	
0	176	~	On early versions of the Model 33 and 35, either of these codes may be generated by either the ALT MODE or ESC key.
1	177	DEL	Delete, rub out. Repeats on Model 37.

Keys That Generate No Codes

REPT	Model 33 and 35 only: causes any other key that is struck to repeat continuously until REPT is released.
PAPER ADVANCE	Model 37 local line feed.
LOCAL RETURN	Model 37 local carriage return.
LOC LF	Model 33 and 35 local line feed.
LOC CR	Model 33 and 35 local carriage return.
INTERRUPT, BREAK	Opens the line (machine sends a continuous string of null characters).
PROCEED, BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 20-character message.

DATA GENERAL CORPORATION
Southboro, Massachusetts 01772