User's Manual

PROGRAMMER'S REFERENCE

microNOVA COMPUTERS

015-000050-00

Ordering No. 015-000050
© Data General Corporation, 1976
All Rights Reserved
Printed in the United States of America
Revision 00, February 1976

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

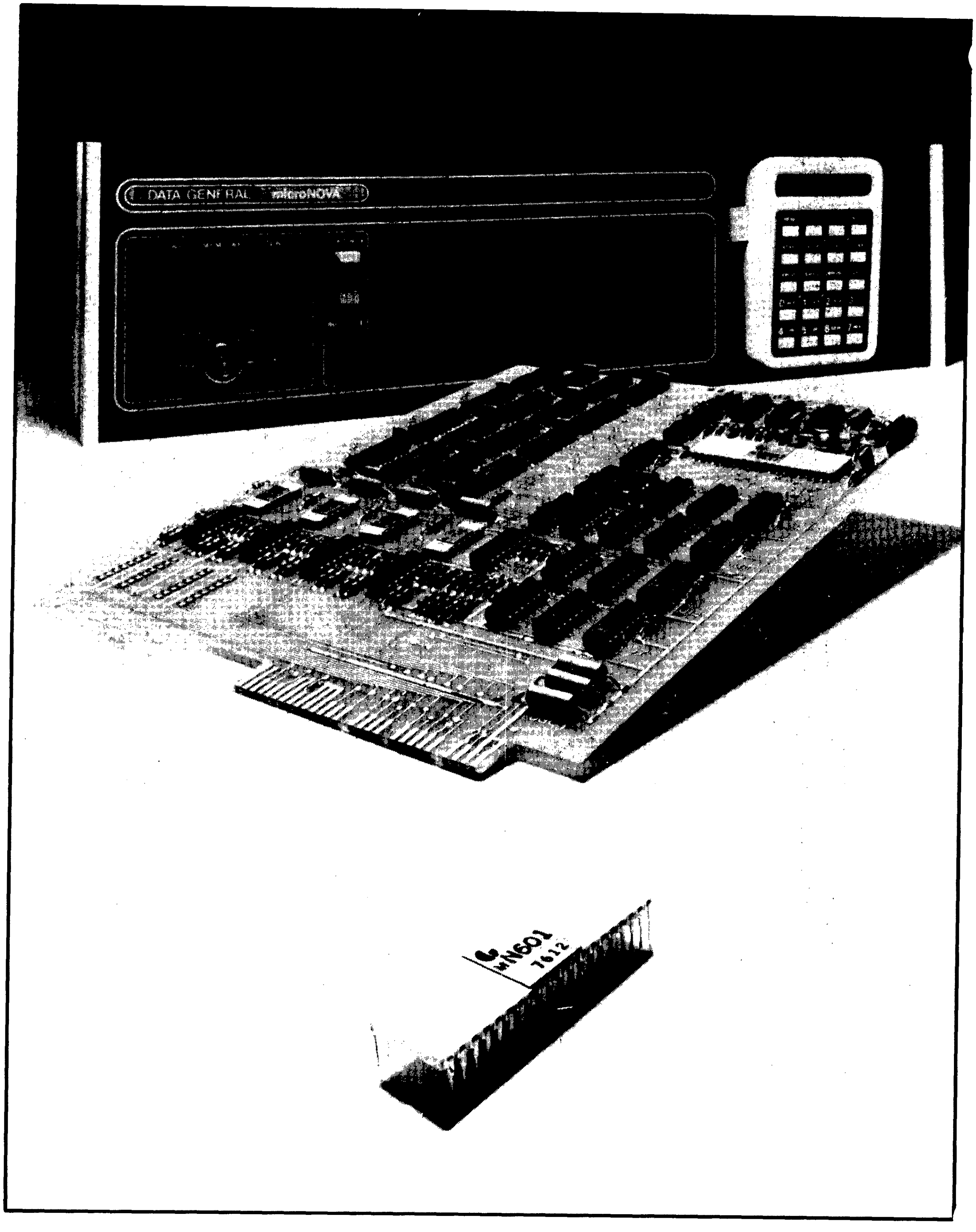
TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

This page intentionally left blank.


DATA GENERAL microNOVA
mN601
7612

SECTION I microNOVA COMPUTERS

INTRODUCTION

The Data General Corporation microNOVA computers are general purpose, four accumulator, stored-program computers with a word length of 16 bits. They have the capability to address up to 32,768 16-bit words. The accumulators are 16 bits in length and are used for arithmetic and logical operations. Two of these accumulators can also be used as index registers. Memory can be addressed either directly or by using indirect addresses. Chains of indirect addresses can be up to eight levels deep. A direct memory access (DMA) data channel is provided to enable rapid data transfer between main memory and peripheral devices. The features of the microNOVA computers are summarized below.

Instruction Set

The basic instruction set for the microNOVA computers contains instructions that perform fixed point arithmetic between accumulators, including multiply and divide; transfer of operands between accumulators and main memory; logical operations between accumulators; transfer of program control; and I/O operations. All instructions are one 16-bit word in length. The arithmetic and logical instructions have the capability to perform, in one instruction, the following sequence: perform an operation, shift the result one bit left or right, test the result of the shift, and then conditionally skip the next instruction depending upon the outcome of the test. In addition, it is possible to perform this entire sequence without affecting either of the operands. This means that complicated numerical manipulation and testing can be performed using a small number of instructions.

The instruction set for the microNOVA computers contains the instruction set for the NOVA line of computers. The multiply and divide instructions which are optional with the NOVA line of computers are standard with the microNOVA computers. In addition, the stack facility which is standard with the NOVA 3 computers is also standard with the microNOVA computers.

Even though the mnemonics and functions performed are the same for all instructions in both instruction sets, the instruction operation codes are different for two of the instructions (I/O RESET and READ SWITCHES). Programs written for NOVA line computers need only be reassembled before they can be run on microNOVA computers.

Multiply/Divide

The multiply and divide instructions allow the multiplication and division of operands to be performed quickly, without resorting to time-consuming software routines. Two 16-bit operands can be multiplied together to yield a 32-bit result. A 16-bit operand can be divided into a 32-bit operand to yield a 16-bit quotient and a 16-bit remainder.

Stack

A last-in/first-out (LIFO) or push-down stack is maintained by the processor. This feature provides a convenient method for the saving of return information and passing arguments between subroutines. The stack also provides an expandable area for the temporary storage of variables and intermediate results.

Memory

Memory is available for microNOVA computers in several different forms and amounts. Semiconductor random-access memory (RAM) is available in modules of either 4 or 8K 16-bit words. Semiconductor programmable read-only memory (PROM) is available in modules of 512, 1K, 2K and 4K 16-bit words.

One of the available I/O devices for the microNOVA is a PROM programmer. This PROM programmer allows programming of PROM's to be an online process one memory module at a time instead of on a chip-by-chip basis.

Power Fail/Auto-restart

The power fail/auto-restart feature of the microNOVA computers provides a "fail-soft" capability in the event of unexpected power loss. In the event of power failure, there is a delay of one to two milliseconds before the processor shuts down. The power fail portion of the feature senses the imminent loss of power and interrupts the processor. The interrupt service routine can then use this delay to store the contents of the accumulators, the program restart address, and other information that will be needed to restart the system. One to two milliseconds is enough time to execute 1,000 to 1,500 instructions on microNOVA computers so there is more than enough time to perform the power fail routine.

When power is restored, the action taken by the auto-restart portion of the feature depends upon the position of the power switch on the front panel. If the switch is in the "run" position, the processor remains stopped after power is restored.

If the switch is in the "lock" position, 50 milliseconds after power is restored, the processor executes the instruction contained in a pre-defined location of main memory, restarting the interrupted system.

The battery backup option available with the microNOVA computers operates in conjunction with the power fail/auto-restart feature to preserve the contents of semiconductor RAM in the event of a power failure. If power fails, the battery backup option will supply power to the memories for a period of up to 45 minutes so that they will not lose their data. An external battery backup option is available which enables the customer to connect larger batteries and thus extend the period of time during which the integrity of the memories can be maintained

Input/Output Bus

The input/output (I/O) bus is that portion of the computer system that carries commands and data between the processor and the various peripheral devices in the system. The I/O bus of the microNOVA computers is a 2-bit parallel, bidirectional, differential data bus. The bus consists of 1 differential I/O clock, 1 differential master clock, 2 differential data paths, a clear line, an interrupt request line, and a data channel request line. To the programmer, the bus appears to be made up of a device selection network, interrupt circuitry, command circuitry, and a 16-bit wide data path.

Device Addressability

Each I/O device in the system has a unique 6-bit device code. Each device is connected to the device selection network in such a way that it will only respond to commands that contain its own device code. The fact that the selection network uses 6-bit device codes gives $2^6 = 64$ unique device codes. Three of these codes are reserved for specific features and functions, but there are still 61 device codes available for use with I/O devices.

Interrupt Capability

The interrupt circuitry contained in the I/O bus provides the capability for any I/O device to interrupt the system when that device requires service. When a device requests an interrupt, the processor automatically transfers program control to the main interrupt service routine. This routine can either poll all the I/O devices in the system to find out which one initiated the interrupt or use a special instruction to identify the source of the interrupt.

The interrupt circuitry of microNOVA computers also contains the capability to implement up to sixteen levels of priority interrupts. This is done with a 16-bit priority mask. Each level of device priority is associated with a bit in this mask. In order to suppress interrupts from any priority level, the corresponding bit in the mask is set to 1.

Data Channel

Handling data transfers between external devices and memory under program control requires an interrupt plus the execution of several instructions for each word transferred. To allow the block transfer of data, the I/O bus contains circuitry for a direct memory access (DMA) data channel though which a device, at its own request, can gain direct access to memory using a minimum of processor time. At the maximum input rate of approximately 150,000 words per second and at the maximum output rate of approximately 172,000 words per second, the data channel effectively stops the processor, but at lower rates processing continues while data is being transferred.

Ease of Interfacing

Due to the straightfoward logic and general design of the I/O bus on the microNOVA computers, and the extensive interface aids offered by Data General, customer provided or customer designed I/O devices may be interfaced easily to a microNOVA computer system.

Input/Output Devices

Data General offers several standard I/O devices for the microNOVA computers. A dual diskette subsystem gives the capability for online random-access data storage. A hand-held console gives the capability for examining and modifying the accumulators and main memory and for controlling the actions of the computer. An asynchronous interface allows either a teletypewriter or video display terminal to be used as a system console device. An option available with the asynchronous interface is an interactive debugger implemented in 256 16-bit words of ROM/RAM on the interface board. This option allows troubleshooting of applications programs directly from the system console without giving up memory space to a software debugger. A PROM programmer allows the customer to program his PROM's as an online process.

Real-time Clock

The real-time clock feature of the microNOVA computers provides a facility for periodic interrupts. When enabled, the clock will interrupt the processor every 2.4 milliseconds. Real-time clock interrupts cause the processor to transfer control to a location different from the location used for other I/O interrupts.

Software

MicroNOVA computers are fully supported by proven Data General software. Because microNOVA computers are compatible with the NOVA line of computers, many of the programming systems available with the NOVA line of computers are also available with microNOVA computers.

Languages

In addition to an editor, macro assembler, relocatable loader, and symbolic debugger, a FORTRAN IV compiler with real-time extensions is available with microNOVA computers. All the standard library routines for arithmetic operations, string manipulation, and input/output operations are included to ease the job of implementing applications systems.

Operating Systems

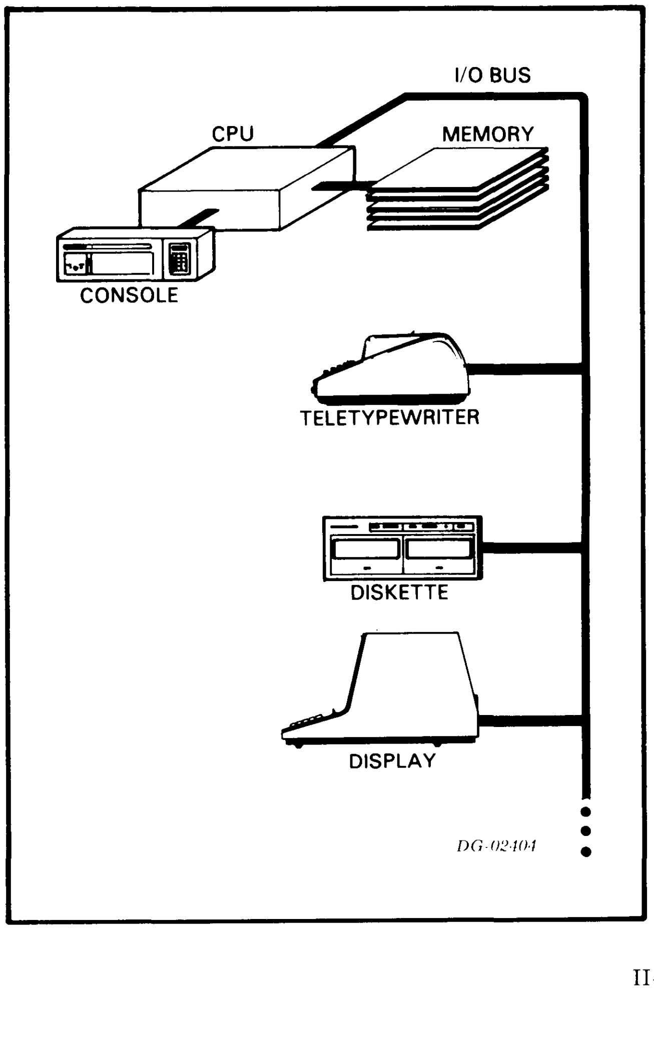
Two operating systems are available for systems using microNOVA computers. The diskette based Disc Operating System (DOS) is a subset of Data General's Real-time Disc Operating System (RDOS). For those applications requiring a small, memory based system, Data General's Real-time Operating System (RTOS) will efficiently manage system resources in a real-time environment.

This page intentionally left blank.

SECTION II
INTERNAL STRUCTURE

INTRODUCTION

The basic structure of a microNOVA data processing system consists of a central processing unit (CPU), some amount of main memory, the I/O bus, the I/O devices connected to the I/O bus, and a console.



The type, size, and number of memory modules and I/O devices have no effect upon the internal logical structure of the CPU. This chapter deals with the addressing of information and the logical representation of information within the CPU, and is unaffected by those portions of the system outside the CPU.

INFORMATION FORMATS

The basic piece of information within the processor is the binary digit, or "bit". A bit is capable of representing only two quantities 0 and 1. However, a bit cannot represent both these values at the same time. At any one point in time, a bit can either represent a 0 or a 1, never both.

The normal unit of information within the CPU is the "word". A word is made up of sixteen bits. Because each bit is capable of representing two quantities, a word is capable of representing $2^{16} = 65{,}536$ different quantities. A word may be divided into two "bytes" of 8-bits each. A byte is capable of representing $2^8 = 256$ different quantities. I/O devices transfer information in units of bits, bytes, words, or groups of words called "records", depending upon the device.

Bit Numbering

In order to avoid confusion when talking about the information contained in bytes and words, the bits that make up these units of information are numbered from left to right, with the leftmost (high-order) bit always numbered bit 0. The numbering extends to the right and is always carried out in the decimal number system. The rightmost (low-order) bit in a byte is bit 7. The rightmost bit in a word is bit 15.

Octal Representation

Because talking about the binary data contained in bytes and words would quickly become awkward and confusing if each bit were described, the octal representation of binary information will be used in this manual. To convert a piece of binary information to its octal representation, the bits in the quantity are separated into groups of three bits each, starting from the right and proceeding to the left. If the number of bits to be represented is not evenly divisible into groups of three, the leftmost group will contain one or two bits. Each group of bits can now be represented by one of eight different symbols. The digits 0-7 are used to represent the quantities 0-7. Each encoded digit is called an octal digit. Because each group of bits can contain any one of 8 values, this representation is somtimes called "base 8" representation.

Another way to represent binary information is the hexadecimal or "hex" representation. In hexadecimal, the bits in the quantity are separated into groups of four bits each and each group can be represented by one of 16 different symbols. The digits 0-9 are used to represent the quantities 0-9. The letters A-F are used to represent the quantities 10-15. Because each group of bits can contain any one of 16 values, this representation is sometimes called "base 16" representation.

The following table gives the correspondence between the various representations.

DECIMAL	BINARY	HEX	BINARY	OCTAL
0	0000	0	000	0
1	0001	1	001	1
2	0010	2	010	2
3	0011	3	011	3
4	0100	4	100	4
5	0101	5	101	5
6	0110	6	110	6
7	0111	7	111	7
8	1000	8	1 000	10
9	1001	9	1 001	11
10	1010	A	1 010	12
11	1011	B	1 011	13
12	1100	C	1 100	14
13	1101	D	1 101	15
14	1110	E	1 110	16
15	1111	F	1 111	17

Our normal decimal numbering system is sometimes called "base 10" representation. Because it is sometimes possible to confuse numbers written in hex or octal with those written in decimal, a subscript denoting the base will be used in cases where confusion might occur. The following examples illustrate this convention.

$$64_{10} = 40_{16} = 100_8$$
$$87_{10} = 57_{16} = 127_8$$
$$63_{10} = 3F_{16} = 77_8$$

In the last example, it is obvious that 3F is a number written in hex, but the subscript is included to erase any possible doubts.

Conversion tables for hex to decimal and octal to decimal are contained in Appendix B of this manual.

Character Codes

Within the processor, all information is represented by binary quantities. The CPU does not recognize certain bit combinations as characters and certain other bit combinations as numbers. Sooner or later, however, this information must be transferred outside the computer in some form easily understood by humans. For this reason, some standard correspondence must be made between certain bit combinations and printable symbols. The code used to implement this correspondence in I/O devices available with microNOVA computers is called the American Standard Code for Information Interchange (ASCII). This code can represent 95 printable symbols plus 33 control functions. A complete table of codes and their corresponding characters can be found in Appendix C of this manual.

Information Representation

Even though the CPU does not intrinsically recognize one information type from another, the different instructions in the instruction set expect that the information to be operated on will be in a specific format. In general, there are four different, basic information formats. They are integers, floating point numbers, logical quantities, and decimal numbers.

Integers

Integers can be represented as either signed or unsigned numbers and can be carried in either single or multiple precision. Single precision integers are two bytes long, while multiple precision integers are four or more bytes long. Unsigned integers use all the available bits to represent the magnitude of the number. A single two-byte word can represent any unsigned number in the inclusive range 0 to 65,535. Two words taken together as an unsigned, double precision integer can represent any number in the inclusive range 0 to 4,294,967,295.

For signed operations, the two's complement numbering system is used. In this system, the leftmost or high-order bit is used as a sign bit. If the sign bit is 0, the number is positive and the remainder of the bits in the number represent the magnitude of the number as described above. If the sign bit is 1, the number is negative and the remainder of the bits represent the two's complement of the magnitude of the number.

To create the negative of a number in the two's complement scheme, complement all the bits of the number including the sign bit. After the complementing process is finished, add 1 to the rightmost or low-order bit. If the two's complement of a negative number is formed, the result will be the corresponding positive number. There is only one representation for zero in two's complement arithmetic: it is the number with all bits zero. Forming the two's complement of zero will produce a carry out of the high-order bit and leave the number with all bits zero.

Examples:

```
To form the negative of 4:
               4 - 0  000  000  000  000  100
      complement - 1  111  111  111  111  011
      add 1      +                          1
                 ----------------------------
               4 - 1  111  111  111  111  100
To form the negative of 1715₈
                 - 0  000  001  111  001  101
      complement - 1  111  110  000  110  010
      add 1      +                          1
                 ----------------------------
                 - 1  111  110  000  110  011
To form the negative of -1715₈
                 - 1  111  110  000  110  011
      complement - 1  000  001  111  001  100
      add 1      +                          1
                 ----------------------------
                 - 0  000  001  111  001  101
To form the negative of 0₈
               0 - 0  000  000  000  000  000
      complement - 1  111  111  111  111  111
      add 1      +                          1
                 ----------------------------
               0 - 0  000  000  000  000  000
```

Note that 0 is a positive number, i.e., its sign bit is 0.

Because the two's complement scheme has only one representation for 0, there is always one more negative number than there are non-negative numbers. The most negative number is a number with a 1 in the sign bit and all other bits 0. The positive value of this number can not be represented in the same number of bits as used to represent the negative number.

A single two-byte word can represent any signed number in the inclusive range -32,768 to +32,767. Two words taken together as a signed, double precision integer can represent any number in the inclusive range -2,147,483,648 to +2,147,483,647.

It is one property of numbers using the two's complement scheme that addition and subtraction of signed numbers are identical to addition and subtraction of unsigned numbers. The CPU just treats the sign bit as the most significant magnitude bit.

Floating Point Numbers

Floating point numbers allow operations to be performed on signed numbers having a much larger range than those normally represented as integers. It would take a 16-word multiple precision integer to represent the range of a floating point number in the microNOVA format. Since floating point numbers occupy either two words for single precision or four words for double precision, floating point arithmetic is used when numbers having a large range must be manipulated.

A floating point number is made up of three parts: the sign, the exponent, and the mantissa. The value of a floating point number is defined to be:

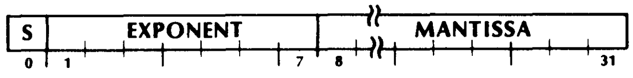
(MANTISSA) X (16 *-RAISED TO THE TRUE VALUE OF THE EXPONENT FIELD*)

The number is signed according to the value of the sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

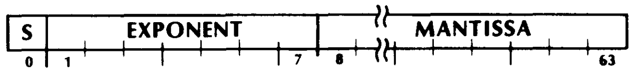
Floating point numbers are represented internally by either 32 bits (single precision) or 64 bits (double precision).

The formats are shown below:

Single Precision



Double Precision



Bit zero is the sign bit: 0 for positive, 1 for negative.

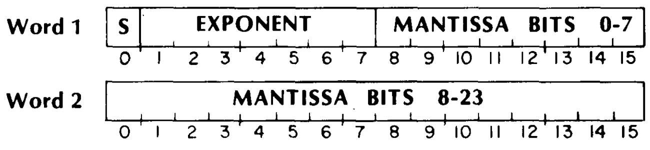
Bits 1-7 contain the exponent. This is the power to which 16 must be raised in order to give the correct value to the number. So that the exponent field may accommodate a large range, "Excess 64" representation is used. This means that the value in the exponent field is 64 greater than the true value of the exponent. If the exponent field is zero, the true value of the exponent is -64. If the exponent field is 64, the true value of the exponent is 0. If the exponent field is 127, the true value of the exponent is 63.

Bits 8-31 for single precision and bits 8-63 for double precision contain the mantissa. This means that bit 8 of the floating point number is bit 0 of the mantissa. The mantissa is always a positive fraction greater than or equal to 1/16 and less than 1. The "binary point" can be thought of as being just to the left of bit 8. Continuing this concept then, bit 8 represents the value 1/2, bit 9 represents the value 1/4, bit 10 represents the value 1/8, and so on.

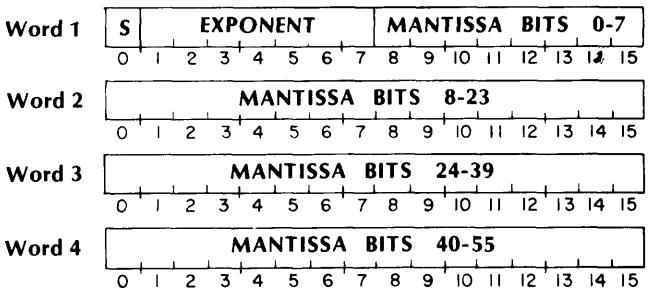
In order to keep the mantissa in the range of 1/16 to 1, the results of floating point arithmetic are "normalized". Normalization is the process whereby the mantissa is shifted left one hex digit at a time until the high-order four bits represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one. Since the mantissa is shifted four bits at a time, it is possible for the high-order three bits of a normalized mantissa to be zero.

Zero is represented by a floating point number with all bits zero. This is true for both single and double precision. This is known as "true zero".

Floating point operands in memory are represented by two words for single precision and by four words for double precision. The formats are shown below:



Double Precision



Logical Quantities

Logical operations in the microNOVA computers can be performed upon individual words. When using the logical operations, the words operated on are treated as unstructured binary quantities.

Decimal Numbers

Decimal numbers may be represented internally in two ways, unpacked decimal and packed decimal. In unpacked decimal, the number is made up of a string of ASCII characters and the sign, if present, may appear in one of four places. The sign of the number may be indicated by a leading or trailing byte which contains the ASCII code for plus ($2B_{16}$) or minus ($2D_{16}$). Alternatively, either the high-order digit or the low-order digit of the number may indicate the sign in addition to carrying a digit of the number. The table below gives the correspondence between certain ASCII characters and the sign and digit values that they carry.

SIGN VALUE	DIGIT VALUE	ASCII CHARACTER	HEX CODE
+	0	b̸ + 0 {	20, 2B, 30, 7B
+	1	1 A	31, 41
+	2	2 B	32, 42
+	3	3 C	33, 43
+	4	4 D	34, 44
+	5	5 E	35, 45
+	6	6 F	36, 46
+	7	7 G	37, 47
+	8	8 H	38, 48
+	9	9 I	39, 49
-	0	- }	2D, 7D
-	1	J	4A
-	2	K	4B
-	3	L	4C
-	4	M	4D
-	5	N	4E
-	6	O	4F
-	7	P	50
-	8	Q	51
-	9	R	52

The digits that are not carrying the sign must be valid ASCII characters for the digits 0-9 (30_{16} - 39_{16}) or spaces (20_{16}). A space has the same value as a zero.

Examples:

In the following examples, the hex value of a byte is shown inside the box; the corresponding ASCII character is shown beneath the box.

	Byte	Byte	Byte	Byte	Byte
+2,048 (leading sign)	2B	32	30	34	38
	+	2	0	4	8
-1,756 (trailing sign)	31	37	35	36	2D
	1	7	5	6	-
+1,850 (high-order sign)	41	38	35	60	
	A	8	5	0	
-3,970 (low-order sign)	33	39	37	7D	
	3	9	7	}	

For packed decimal, each digit of the decimal number occupies one hex digit. The sign is specified by a trailing hex digit. The number must start and end on a byte boundary. In other words, the number cannot start or end halfway through a byte. This means that a packed decimal number will always consist of an odd number of digits followed by the sign. The sign must be either C_{16} for plus or D_{16} for minus. The only valid codes for digits are $0\text{-}9_{10}$.

Examples:

In the following examples, the hex value of a digit is shown within the box; the corresponding decimal digit is shown beneath the box.

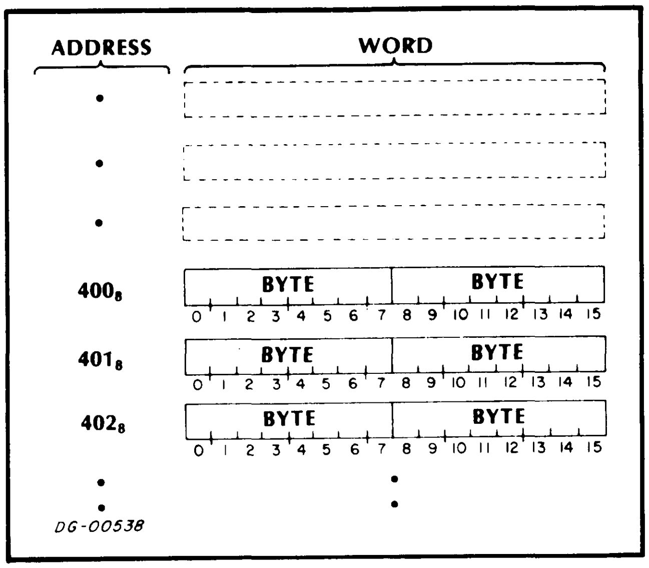
	Byte		Byte		Byte	
+2,048	0	2	0	4	8	C
	0	2	0	4	8	+
+32,456	3	2	4	5	6	C
	3	2	4	5	6	+
-1,756	0	1	7	5	6	D
	0	1	7	6	6	-
-25,989	2	5	9	8	9	D
	2	5	9	8	9	-

INFORMATION ADDRESSING

The information formats described in the preceding section give a way of representing different types of data within the CPU. Operations cannot be performed upon these data types, however, unless they can be addressed by the CPU. The address of a piece of information is its location in main memory. Once the CPU knows the address of a piece of information, the desired operation can be performed.

Word Addressing

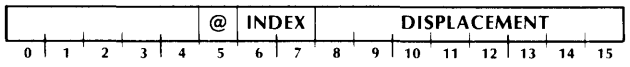
Main memory is partitioned into 2-byte words, and each word has an address. The first word in memory has the address 0. The next word has the address 1, the next word has the address 2, and so on. Word addressing is used to address integers, floating point numbers, and logical quantities that are formatted in units of words.



Effective Address Calculation

There are six instructions in the microNOVA instruction set that directly reference memory using word addressing. These instructions use 11 bits in the instruction word to define the address. These bits do not directly specify the address, but are used in a calculation which results in the address of the desired word. The resultant address is called the "effective address" or "E", and the calculation is called the "effective address calculation".

The 11 bits in the instruction that are used to define the effective address are bits 5-15. Bit 5 is called the "indirect bit", bits 6 and 7 are called the "index bits" and bits 8-15 are called the "displacement bits"



If the index bits are 00, the displacement bits are treated as an unsigned number which is the address of a word in memory. This is called "absolute addressing". Absolute addressing can be used to directly address any of the first 256 words in main memory.

If the index bits are 01, the displacement bits are treated as a signed, two's complement number which is added to the address of the instruction. This is called "relative addressing". Relative addressing can be used to directly address any word in main memory whose address is in the range -128 to +127 from the instruction.

If the index bits are 10, accumulator 2 is used as an index register. if the index bits are 11, accumulator 3 is used as an index register. In this form of word addressing, known as "index register addressing" the displacement is treated as a signed, two's complement number which is added to the contents of the selected index register to produce a memory address. In index register addressing, the addition of the displacement to the contents of the index register does not change the value contained in the index register. Index register addressing can be used to directly reference any word in main memory.

The result of the addition performed in relative addressing and index register addressing is "clipped" to 15 bits. In other words, the high-order bit of the result is set to 0. For example, if accumulator 2 is to be used as an index register and contains the number 077774_8, and the displacement bits contain the number 012_8, then the result of the addition would be 000006_8, not 1000006_8.

After one of the three types of addresses has been computed from the index and displacement bits, the indirect bit is tested. If this bit is 0, the address already computed is taken as the effective address. If the indirect bit is 1, the word addressed by the result of the index and displacement bits is assumed to contain an address. The word at this intermediate address is retrieved. In this word, bit 0 is the indirect bit and bits 1-15 contain an address. If bit 0 of the referenced word is 1, another level of indirection is indicated, and bits 1-15 contain the address of the next word in the indirection chain. The processor will continue to follow this chain of indirect addresses by retrieving words until a word is retrieved with bit 0 set to 0. When a word is retrieved with bit 0 set to 0, bits 1-15 of this word are taken to be the effective address.

In order to protect against indirection chains that never end, the processor uses an internal counter to count the number of levels of indirection it has followed. This counter is set to 0 if the indirect bit in the instruction is 1. Each time a word in the chain is retrieved, 2 is added to the counter. If the counter becomes greater than 15 before a word is retrieved with bit 0 set to 0, the processor executes a HALT instruction. In this case, the instruction is not completed. The contents of all accumulators and the carry bit remain unchanged. The program counter addresses the word following the uncompleted instruction unless that instruction was a JUMP or JUMP TO SUBROUTINE instruction, in which case the contents of the program counter are unpredictable.

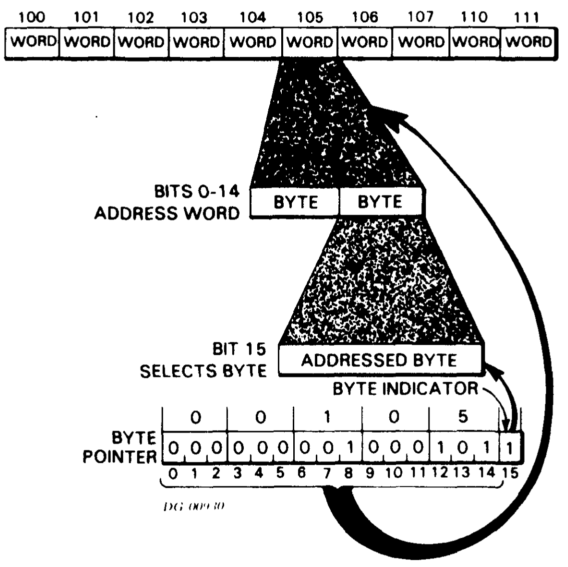
If an indirect address points to a location in the range $20\text{-}27_8$ (auto-increment locations), that word is fetched, the contents of the word are incremented by one and written back into the location. This updated value is then used to continue the addressing chain. If an indirect address points to a location in the range $30\text{-}37_8$ (auto-decrement locations), that word is fetched, the contents of the word are decremented by one and written back into the location. The updated value is then used to continue the addressing chain. Each indirect reference of an auto-increment or auto-decrement location increments the internal counter an extra time.

NOTE *When referencing auto-increment and auto-decrement locations, the state of bit 0 before the increment or decrement is the condition upon which the continuation of the indirection chain is based. For example: If an auto-increment location contains 177777_8, and the location is referenced as part of an indirection chain, location 0 will be the next address in the chain. That is, the effective address will not be 0.*

An effective address is always 15 bits in length. This means that an instruction which uses the effective address calculation can address any of $32{,}768_{10}$ words. This gives rise to the concept of an "address space", which, in the microNOVA computer, contains 64K bytes or 32,768 2-byte words.

Byte Addressing

While bytes in main memory cannot be directly addressed by the CPU, there is a convenient programming method for manipulating individual bytes of information. This technique involves the use of a "byte pointer". A byte pointer is a word in which bits 0-14 are the address in memory of a 2-byte word. Bit 15 of the byte pointer is the "byte indicator". If the byte indicator is 0, the referenced byte is the high-order (bits 0-7) byte of the word addressed by byte pointer bits 0-14. If the byte indicator is 1, the referenced byte is the low-order (bits 8-15) byte of the word addressed by byte pointer bits 0-14.

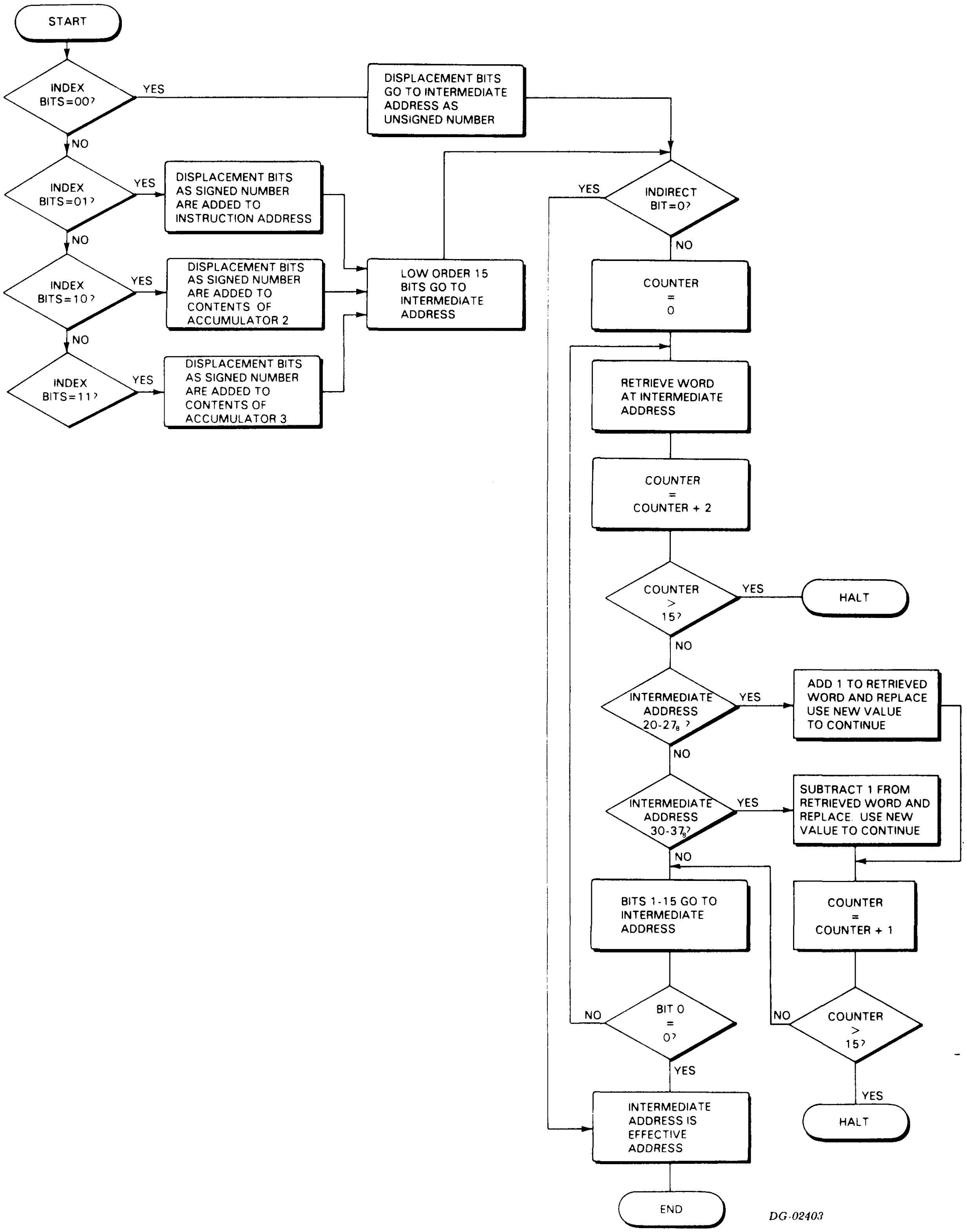


Programming routines to load and store individual bytes using byte pointers are given in Appendix E of this manual.

Addressing Nonexistent Memory

The address space of a microNOVA computer contains 32K 16-bit words. This means that the CPU can address 32,768 separate memory locations. It is possible, however, that some of these addresses will not have physical memory locations associated with them. If an attempt is made to retrieve a word from a memory location that does not exist, the CPU functions as if the location exists and has all its bits set to 1. If an attempt is made to write a word into a memory location that does not exist, the CPU functions as if the location does exist and no indication is given that it does not exist.

EFFECTIVE ADDRESS CALCULATION


START
INDEX BITS=00?
YES
NO
DISPLACEMENT BITS GO TO INTERMEDIATE ADDRESS AS UNSIGNED NUMBER
INDEX BITS=01?
YES
NO
DISPLACEMENT BITS AS SIGNED NUMBER ARE ADDED TO INSTRUCTION ADDRESS
INDEX BITS=10?
YES
NO
DISPLACEMENT BITS AS SIGNED NUMBER ARE ADDED TO CONTENTS OF ACCUMULATOR 2
INDEX BITS=11?
YES
DISPLACEMENT BITS AS SIGNED NUMBER ARE ADDED TO CONTENTS OF ACCUMULATOR 3
LOW ORDER 15 BITS GO TO INTERMEDIATE ADDRESS
INDIRECT BIT=0?
YES
NO
COUNTER = 0
RETRIEVE WORD AT INTERMEDIATE ADDRESS
COUNTER = COUNTER + 2
COUNTER > 15?
YES
HALT
NO
INTERMEDIATE ADDRESS 20-27₈ ?
YES
ADD 1 TO RETRIEVED WORD AND REPLACE USE NEW VALUE TO CONTINUE
NO
INTERMEDIATE ADDRESS 30-37₈?
YES
SUBTRACT 1 FROM RETRIEVED WORD AND REPLACE. USE NEW VALUE TO CONTINUE
NO
BITS 1-15 GO TO INTERMEDIATE ADDRESS
COUNTER = COUNTER + 1
BIT 0 = 0?
NO
YES
COUNTER > 15?
NO
YES
HALT
INTERMEDIATE ADDRESS IS EFFECTIVE ADDRESS
END
DG-02403

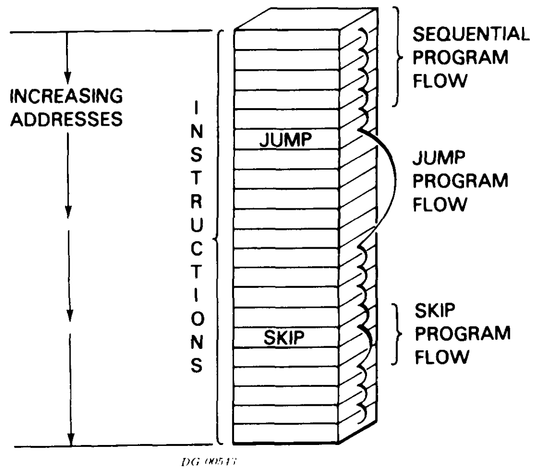
PROGRAM EXECUTION

Programs for microNOVA computers consist of sequences of instructions that reside in main memory. The order in which these instructions are executed depends on a 15-bit counter called the "program counter". The program counter always contains the address of the instruction currently being executed. After the completion of each instruction the program counter is incremented by one and the next instruction is fetched from that address. This method of operation is called "sequential operation" and the instruction fetched from the location addressed by the incremented program counter is called the "next sequential instruction".

Program Flow Alteration

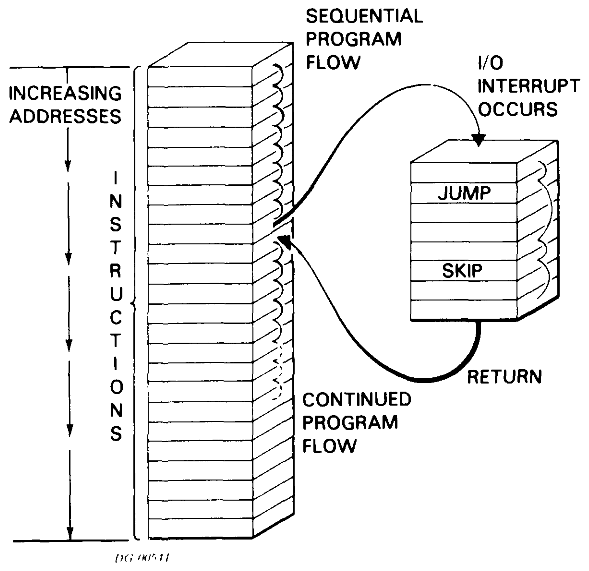
Sequential operation can be explicitly altered by the programmer in two ways. Jump instructions alter program flow by inserting a new value into the program counter. Conditional skip instructions can alter program flow by incrementing the program counter an extra time if a specified test condition is true. In the case of a conditional skip instruction when the test condition is true, the next sequential instruction is not executed because it is not addressed. After either a jump instruction or a successful conditional skip instruction, sequential operation continues with the instruction addressed by the updated value of the program counter.

Because the program counter is 15 bits in length, it can address 32,768 separate memory locations. The next memory location after 77777_8 is location 0, and the location before 0 is location 77777_8. If the program counter rolls from 77777_8 to 0 in the course of sequential operation, no indication is given and processing continues with the location addressed by the updated value of the program counter.



Program Flow Interruption

The normal flow of a program may be interrupted by external or exceptional conditions such as I/O interrupts or various kinds of faults. In these cases, the address of the next sequential instruction in the interrupted program is saved by the CPU so that the I/O handler or the various fault handlers can return control to the program at the correct point. Once the address of the next sequential instruction in the program has been placed in the program counter by the I/O handler or the fault handler, sequential operation of the program resumes.



SECTION III
INSTRUCTION SETS

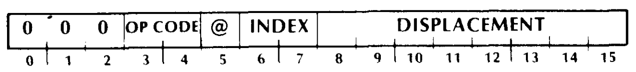
INTRODUCTION

The instruction set implemented on the microNOVA computers is divided into 5 instruction sets. There are instruction sets available for fixed point arithmetic, logical operations, stack manipulation, program flow alteration, and I/O operations. In addition, instruction sets are available for programming the hand-held console and certain CPU functions.

INSTRUCTION FORMATS

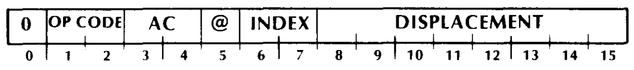
There are four different formats for instructions on the microNOVA computers. These formats allow an extensive instruction set while still keeping the instruction length to one word. The four formats and their general layouts are described below.

No Accumulator-Effective Address



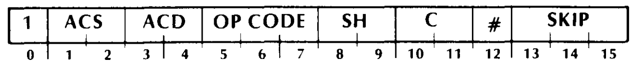
In the No Accumulator-Effective Address format instructions, bits 0-2 are 000, and bits 3-4 contain the operation code. The effective address is computed from bits 5-15 as described under "Effective Address Calculation".

One Acumulator-Effective Address

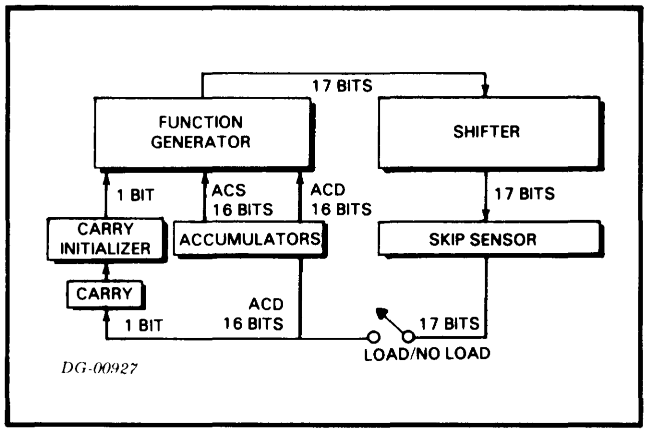


In the One Accumulator-Effective Address format instructions, bit 0 is 0 and bits 1-2 contain the operation code. Bits 3-4 specify the accumulator for the opearation. The effective address is computed from bits 5-15 as described under "Effective Address Calculation".

Two Accumulator-Multiple Operation



In the Two Accumulator-Multiple Operation format instructions, bit 0 is 1, bits 1 and 2 specify the source accumulator, bits 3 and 4 specify the destination accumulator, bits 5-7 contain the operation code, bits 8 and 9 specify the action of the shifter, bits 10 and 11 specify the value to which the carry bit will be initialized, bit 12 specifies whether or not the result will be loaded into the destination accumulator, and bits 13-15 specify the skip test. All instructions in this format utilize an arithmetic unit whose logical organization is illustrated as follows.



Each instruction specifies two accumulators to supply operands to the function generator, which performs the function specified by bits 5-7 of the instruction. The function generator also produces a carry bit whose value depends upon three quantities: an initial value specified by the instruction, the inputs, and the function performed. The initial value may be derived from the previous value of the carry bit or the instruction may specify an independent value.

The 17-bit output of the function generator, made up of the carry bit and the 16-bit function result, then goes to the shifter. In the shifter, the 17-bit result can be rotated one place right or left, or the two 8-bit halves of the function result can be swapped without affecting the carry bit. The 17-bit output of the shifter can then be tested for a skip. The skip sensor can test whether the carry bit or the rest of the 17-bit result is or is not equal to zero. After the skip sensor has tested the shifter output, it can be loaded into the carry bit and the destination accumulator. Note, however, that loading is not necessary. An instruction in this format can perform a complicated arithmetic and shifting operation and test the result for a skip without affecting the carry bit or either of the operands.

Input/Output

0	1	1	AC		OP CODE			CONTROL		DEVICE CODE					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

In the Input/Output format instructions, bits 0-2 are 011, bits 3-4 specify the accumulator for the operation, bits 5-7 contain the operation code, bits 8-9 specify the control signal to be used, and bits 10-15 contain the device code of the referenced device.

CODING AIDS

In the descriptions of the separate instructions, the general form of how the instruction is coded in assembly language is given along with the instruction format and the description of the instruction. The general form of how an instruction may be coded has the following format:

MNEMONIC *[optional mnemonics]* **OPERAND STRING**

The mnemonic must be coded exactly as shown in the instruction description. Some instructions have optional mnemonics that may be appended to the main mnemonic if the option is desired. The operand string is made up of the operands for the given instruction.

Square brackets "[ ]" or "*[ ]*" along with boldface- and italic-printed symbols are used in this manual to aid in defining the instructions. These conventions are used to help describe how an instruction should be written so that it can be recognized by the assembler and translated into the correct binary, or machine language, representation. Their general definition is given below.

[ ], *[ ]*	Square brackets indicate that the enclosed symbol is an optional operand or mnemonic. The operand enclosed in the brackets (e.g., *[,skip]*) may be coded or not, depending on whether or not the associated option is desired.
BOLD	Operands or mnemonics printed in boldface must be coded exactly as shown. For example, the mnemonic for the MOVE instruction is coded **MOV**.
italic	Operands or mnemonics printed in italics require a specific substitution. Replace the symbol with the number of a desired accumulator, or address, or with a user-defined symbol that the assembler recognizes as a specific name, address, number, or mnemonic.

The following abbreviations are used throughout this manual:

AC = Accumulator

ACS = Source Accumulator

ACD = Destination Accumulator

In the instructions that utilize an effective address, the following coding conventions are used:

> *The indirect bit is set to 1 by coding the symbol @ anywhere in the effective address operand string.*
>
> *The "no load" option, available with certain fixed point arithmetic instructions, can be specified by coding the symbol # at the end of the instruction mnemonic, or anywhere in its operand string.*
>
> *The index bits are set by coding a comma followed by one of the digits 0-3 as the last operand of the operand string. The character "period" (.) can be used to set the index bits to 01. "Period" can be read to mean "address of the instruction". When the period is used, it is followed by either a plus or minus sign followed by the displacement e.g., ".+7", or ".-2".*

The displacement is coded as a signed number in the current assembler radix. This radix is the numbering system in which the program supplies numbers to the assembler. The default radix is base 8 or octal. The assembler radix can be changed by using the **.RDX** statement.

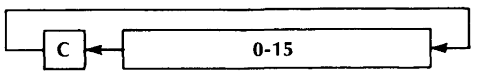
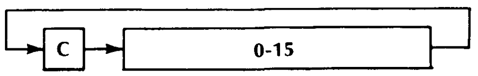
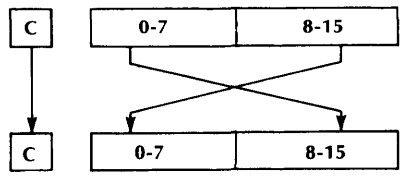
The assembler available with the microNOVA computers allows the programmer to place labels on instructions or locations in memory. When the assembler comes upon a label in the operand string of an effective address instruction, it automatically sets the index and displacement bits to the correct values. For a detailed discussion of the features and operation of the microNOVA assembler, see the assembler manual (*DGC no. 093-000081*).

The fixed point and logical instructions which use the two accumulator-multiple operation format have several options that can be obtained by appending suffixes to the instruction mnemonic and by coding optional operands in the operand string. The characters to be coded are given below with their results.

The characters in the column titled "class abbreviation" refer to specific fields in the two accumulator-multiple operation format. The characters in the column titled "coded character" show the various characters which may be coded for this option. The numbers in the column titled "result bits" show the bit settings in these fields resulting from each coded character. The comments in the column titled "operation" describe the effect of these bit settings.

CLASS ABBREV.	CODED CHARACTER	RESULT BITS	OPERATION
C	(option omitted)	00	Do not initialize the carry bit
	Z	01	Initialize the carry bit to 0
	O	10	Initialize the carry bit to 1
	C	11	Initialize the carry bit to the complement of its present value
SH	(option omitted)	00	Leave the result of the arithmetic or logical operation unaffected
	L	01	Combine the carry and the 16-bit result into a 17-bit number and rotate it one bit left
	R	10	Combine the carry and the 16-bit result into a 17-bit number and rotate it one bit right
	S	11	Exchange the two 8 - bit halves of the 16 - bit result without affecting the carry.
#	(option omitted)	0	Load the result of the shift operation into ACD
	#	1	Do not load the result of the shift operation into ACD.

The following diagrams illustrate the operation of the shifter.

Coded Character	Shifter Operation
L	Left rotate one place. Bit 0 is rotated into the carry position, the carry bit into bit 15. 
R	Right rotate one place. Bit 15 is rotated into the carry position, the carry bit into bit 0. 
S	Swap the halves of the 16 - bit result. The carry bit is not affected. 

The following operands initiate operations that test the result of the shift operation. If the tested condition is true, the next sequential instruction is skipped.

CLASS ABBREV.	CODED CHARACTER	RESULT BITS	OPERATION
SKIP	**(option omitted)**	**000**	**Never skip**
	SKP	**001**	**Always skip**
	SZC	**010**	**Skip if carry = 0**
	SNC	**011**	**Skip if carry ≠ 0**
	SZR	**100**	**Skip if result = 0**
	SNR	**101**	**Skip if result ≠ 0**
	SEZ	**110**	**Skip if either carry or result = 0**
	SBN	**111**	**Skip if both carry and result ≠ 0**

NOTE *Instructions in the Two Accumulator-Multiple Operation format must not have both the "No Load" and the "Never Skip" options specified at the same time. These bit combinations are used by other instructions in the instruction set.*

As an example of how to use these tables, assume that accumulator 3 contains a signed, two's complement number. Now consider the problem of determining whether this number is positive or negative. One way to determine this would be to place the number zero in another accumulator and use the SUBTRACT instruction, but this requires an extra instruction and also destroys the previous contents of the other accumulator. Another way to determine the sign of the number in accumulator 3 is to use the MOVE instruction and the power of the two accumulator-multiple operation format. With the MOVE instruction, the contents of AC3 can be placed in the shifter and shifted one bit to the left. This places the sign bit in the carry bit. The carry bit can then be tested for zero. In order to preserve the number in AC3, the instruction can prevent the output of the shifter from being loaded back into AC3.

The general form of the MOVE instruction is:
MOV *[c][sh][#]   acs,acd[,skip]*

The general bit pattern of the MOVE instruction is:

1	ACS	ACD	0	1	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 14 15

To shift the number in AC3 one bit left without destroying the number, and skip the next sequential instruction if the bit shifted into the carry bit is zero, the following instruction could be coded:

MOVL #   3,3,SZC

This instruction would assemble into the following bit pattern:

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

FIXED POINT ARITHMETIC

The fixed point instruction set performs binary arithmetic on operands in accumulators. The instruction set provides for loading, storing, adding, subtracting, multiplying, dividing, and comparing of fixed point operands.

Load Accumulator

LDA *ac,[@]displacement[,index]*

0	0	1	AC		@	INDEX		DISPLACEMENT							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The word addressed by the effective address, "E", is placed in the specified accumulator. The previous contents of the location addressed by "E" remain unchanged.

Store Accumulator

STA *ac,[@]displacement[,index]*

0	1	0	AC		@	INDEX		DISPLACEMENT							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The contents of the specified accumulator are placed in the word addressed by the effective address, "E". The previous contents of the location addressed by "E" are lost. The contents of the specified accumulator remain unchanged.

Add

ADD *[c][sh][#] acs,acd[,skip]*

1	ACS		ACD		1	1	0	SH		C		#	SKIP		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The carry bit is initialized to the specified value. The unsigned, 16-bit number in ACS is added to the unsigned, 16-bit number in ACD and the result is placed in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result of the shift is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE *If the sum of the two numbers being added is greater than $65{,}535_{10}$, the carry bit is complemented.*

Subtract

SUB *[c][sh][#] acs,acd[,skip]*

1	ACS		ACD		1	0	1	SH		C		#	SKIP		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The carry bit is initialized to its specified value. The unsigned, 16-bit number in ACS is subtracted from the unsigned, 16-bit number in ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. The result of the addition is placed in the shifter. If the operation produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result of the shift is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE *If the number in ACS is less than or equal to the number in ACD, the carry bit is complemented.*

Negate

NEG *[c][sh][#] acs,acd[,skip]*

1	ACS		ACD		0	0	1	SH		C		#	SKIP		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The carry bit is initialized to the specified value. The two's complement of the unsigned, 16-bit number in ACS is placed in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE *If ACS contains 0, the carry bit is complemented.*

Add Complement

ADC *[c][sh][#] acs,acd[,skip]*

1	ACS		ACD		1	0	1	SH		C		#	SKIP		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The carry bit is initialized to the specified value. The logical complement of the unsigned, 16-bit number in ACS is added to the unsigned, 16-bit number in ACD and the result is placed in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The specified shift operation is performed and the result of the shift is loaded into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE *If the number in ACS is less than the number in ACD, the carry bit is complemented.*

Move

MOV *[c][sh][#]* *acs,acd[,skip]*

1	ACS		ACD		0	1	0	SH		C		#	SKIP		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The carry bit is initialized to the specified value. The contents of ACS are placed in the shifter. The specified shift operation is performed and the result of the shift is loaded into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Increment

INC *[c][sh][#]* *acs,acd[,skip]*

1	ACS		ACD		0	1	1	SH		C		#	SKIP		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The carry bit is initialized to the specified value. The unsigned, 16-bit number in ACS is incremented by one and the result is placed in the shifter. If the incrementation produces a carry of 1 out of the high order bit, the carry bit is complemented. The specified shift operation is performed and the result of the shift is loaded into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE *If the number in ACS is 177777_8, the carry bit is complemented.*

Multiply

MUL

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

The unsigned, 16-bit number in AC1 is multiplied by the unsigned, 16-bit number in AC2 to yield an unsigned, 32-bit intermediate result. The unsigned, 16-bit number in AC0 is added to the intermediate result to produce the final result. The final result is an unsigned, 32-bit number and occupies AC0 and AC1. Bit 0 of AC0 is the high-order bit of the result and bit 15 of AC1 is the low-order bit. The contents of AC2 remain unchanged. Because the result is a double-length number, overflow cannot occur.

Divide

DIV

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

The unsigned, 32-bit number contained in AC0 and AC1 is divided by the unsigned, 16-bit number in AC2. Bit 0 of AC0 is the high-order bit of the dividend and bit 15 of AC1 is the low-order bit. The quotient and remainder are unsigned, 16-bit numbers and are placed in AC1 and AC0, respectively. The carry bit is set to 0. The contents of AC2 remain unchanged.

NOTE *Before the divide operation takes place, the number in AC0 is compared to the number in AC2. If the contents of AC0 are greater than or equal to the contents of AC2, an overflow condition is indicated. The carry bit is set to 1, and the operation is terminated. All operands remain unchanged.*

LOGICAL OPERATIONS

The logical instruction set performs logical operations on operands in accumulators. The operands are 16 bits long and are treated as unstructured binary quantities. The logical operations included in this set are COMPLEMENT and AND.

Complement

COM *[c][sh][#]   acs,acd[,skip]*

1	ACS	ACD	0	0	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 14 15

The carry bit is initialized to the specified value. The logical complement of the number in ACS is placed in the shifter. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

And

AND *[c][sh][#]   acs,acd[,skip]*

1	ACS	ACD	1	1	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 14 15

The carry bit is initialized to the specified value. The logical AND of ACS and ACD is placed in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both ACS and ACD is one; otherwise the result bit is 0. The specified shift operation is performed and the result is placed in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

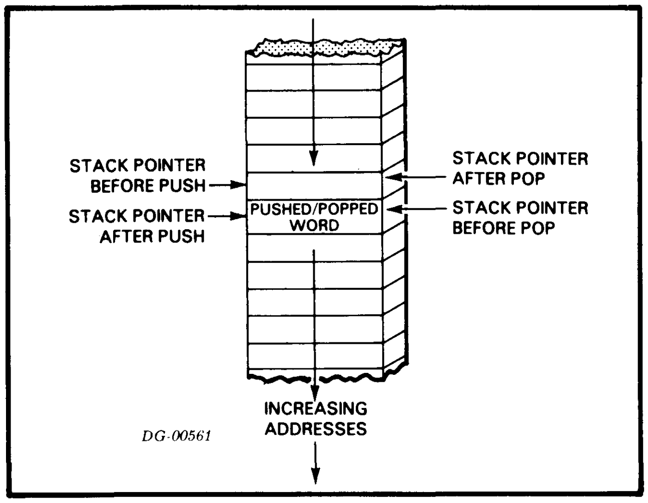
STACK MANIPULATION

An important feature of the microNOVA computers is the stack manipulation facility. A last-in/first-out (LIFO) or "Push-Down" stack is maintained by the processor. The stack facility provides an expandable area of temporary storage for variables, data, return addresses, subroutine arguments, etc. An important byproduct of the stack facility is that storage locations are reserved only when needed. When a procedure is finished with its portion of the stack, those memory locations are reclaimed and are available for use by some other procedure.

The operation of the stack depends upon the contents of two hardware registers. The registers and their contents are described below.

Stack Pointer

The stack pointer is the address of the "top" of the stack and is affected by operations that either "push" objects onto or "pop" objects off of the stack. A push operation increments the stack pointer by 1 and then places the "pushed" object in the word addressed by the new value of the stack pointer. A pop operation takes the word addressed by the current value of the stack pointer and places it in some new location and then decrements the stack pointer by 1.



Frame Pointer

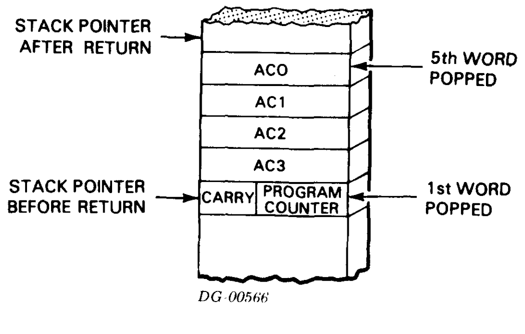
The frame pointer is used to reference an area in the user stack called a "frame". A frame is that portion of the stack which is reserved for use by a certain procedure. The frame pointer usually points to the first available word minus 1 in the current frame. The frame pointer is also used by the RETURN instruction to reset the user stack pointer.

Return Block

A return block is defined as a block of five words that is pushed onto the stack in order to allow a convenient return to the calling program. The format of the return block, therefore, is determined by how it is used in the return sequence. The format of a typical return block is as follows:

WORD No. POPPED	DESTINATION
1	Bit 0 placed in the carry bit. Bits 1 - 15 placed in the program counter
2	AC3
3	AC2
4	AC1
5	AC0

In the stack, the return block looks like this:



Stack Frames

In order to implement re-entrant subroutines, a new area of temporary storage must be available for each execution of a called subroutine. The easiest way to accomplish this is for the subroutine to use the stack for temporary storage. A "stack frame" is defined as that portion of the stack which is available to the called routine. In general, the stack frame belonging to a subroutine begins with the first word in the stack after the return block pushed by the called routine and contains all words in the stack up to, and including, the return block pushed by any routine which the called routine calls. Variables and arguments can be transmitted from the calling routine to the called routine by placing them in prearranged positions in the calling routine's stack frame. Because the SAVE instruction sets the frame pointer to the last word in the return block, these variables and arguments can be referenced by the called program as a negative displacement from the frame pointer. The called routine should ensure that reference to the calling routine's stack frame is made only with the permission of the calling routine.

Stack Protection

During every instruction that pushes data onto the stack, a check is made for stack overflow. If the instruction places data in a word whose address is an integral multiple of 256_{10}, a stack overflow is indicated. If a stack overflow is indicated, the instruction is completed, an internal stack overflow flag is set to 1, and, if the Interrupt On flag is 1, a stack fault is performed. If the Interrupt On flag is 0, the stack overflow flag remains set to 1, and as soon as the interrupt system is enabled, the stack fault is performed.

When a stack fault is performed, the Interrupt On flag is set to 0; the stack overflow flag is set to 0; the updated program counter is stored in memory location 0; and the processor executes a "jump indirect" to memory location 3.

Initialization of the Stack Control Registers

Before the first operation on the stack can be performed, the stack control words must be initialized. The rules for initialization are as follows:

Stack Pointer

The stack pointer must be initialized to the beginning address of the stack area minus one.

Frame Pointer

If the main user program is going to use the frame pointer, it should be initialized to the same value as the stack pointer. Otherwise, the frame pointer can be initialized in a subroutine by the SAVE instruction.

The stack feature of the microNOVA computers is programmed with eight I/O instructions which use the device code 01. Although the instructions are in the standard I/O format, the operation of these instructions is in no way similar to I/O instructions.

Push Accumulator

PSHA *ac*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	1	1	0	0	0	0	0	0	0	1

The contents of the specified accumulator are pushed onto the top of the stack. The contents of the specified accumulator remain unchanged.

Pop Accumulator

POPA *ac*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	1	1	1	0	0	0	0	0	0	1

The specified accumulator is filled with the word popped off the top of the stack.

Save

SAV

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

A return block is pushed onto the stack. After the fifth word of the return block is pushed, the value of the stack pointer is placed in the frame pointer and in AC3. The contents of accumulators 0, 1, and 2 remain unchanged. The format of the five words pushed is as follows:

WORD No. PUSHED	CONTENTS
1	AC0
2	AC1
3	AC2
4	Frame pointer before the SAVE.
5	Bit 0 = carry bit Bits 1 - 15 = bits 1 - 15 of AC3

Move To Stack Pointer

MTSP *ac*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	1	0	0	0	0	0	0	0	0	1

Bits 1-15 of the specified accumulator are placed in the stack pointer. The contents of the specified accumulator remain unchanged.

Move To Frame Pointer

MTFP *ac*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	0	0	0	0	0	0	0	0	0	1

Bits 1-15 of the specified accumulator are placed in the frame pointer. The contents of the specified accumulator remain unchanged.

Move From Stack Pointer

MFSP *ac*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	1	0	1	0	0	0	0	0	0	1

The contents of the stack pointer are placed in bits 1-15 of the specified accumulator. Bit 0 of the specified accumulator is set to 0. The contents of the stack pointer remain unchanged.

Move From Frame Pointer

MFFP *ac*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	0	0	1	0	0	0	0	0	0	1

The contents of the frame pointer are placed in bits 1-15 of the specified accumulator. Bit 0 of the specified accumulator is set to 0. The contents of the frame pointer remain unchanged.

PROGRAM FLOW ALTERATION

As stated previously, the normal method of program execution is sequential. That is, the processor will continue to retrieve instructions from sequentially addressed locations in memory until directed to do otherwise. Instructions are provided in the instruction set that alter this sequential flow. Program flow alteration is accomplished by placing a new value in the program counter. Sequential operation will then continue with the instruction addressed by this new value. Instructions are provided that change the value of the program counter, change the value of the program counter and save a return address, or modify a memory location by incrementing or decrementing and skip the next sequential word if the result is zero.

Jump

JMP *[@]displacement[,index]*

0	0	0	0	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 9 10 11 12 13 14 15

The effective address, "E" is computed and placed in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Jump To Subroutine

JSR *[@]displacement[,index]*

0	0	0	0	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 9 10 11 12 13 14 15

The effective address, "E" is computed. Then the present value of the program counter is incremented by one and the result is placed in AC3. "E" is then placed in the program counter and sequential operation continues with the word addressed by the updated value of the program counter.

NOTE *The computation of "E" is completed before the incremented program counter is placed in AC3.*

Increment And Skip If Zero

ISZ *[@]displacement[,index]*

0	0	0	1	0	@	INDEX	DISPLACEMEMT
0	1	2	3	4	5	6 7	8 9 10 11 12 13 14 15

The word addressed by "E" is incremented by one and the result is written back into that location. If the updated value of the location is zero, the next sequential word is skipped.

Decrement And Skip If Zero

DSZ *[@]displacement[,index]*

0	0	0	1	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 9 10 11 12 13 14 15

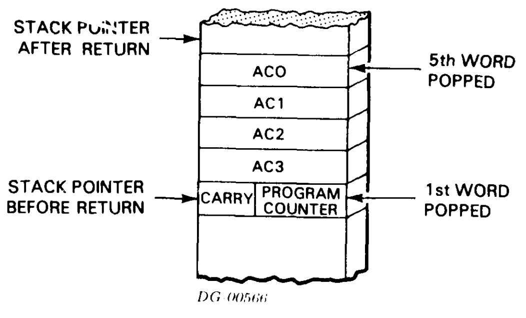
The word addressed by "E" is decremented by one and the result is written back into that location. If the updated value of the location is zero, the next sequential word is skipped.

Return

RET

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

The contents of the frame pointer are placed in the stack pointer and then five words are popped off of the stack and placed in predetermined locations. The words popped and their destinations are as follows:



Sequential operation continues with the word addressed by the updated value of the program counter.

Trap

TRAP *acs,acd,trap number*

1	ACS	ACD	TRAP NUMBER	1 0 0 0
0	1 2	3 4	5 6 7 8 9 10 11	12 13 14 15

The address of this instruction is placed in memory location 46_8 and bit 0 of that location is set to 0. Then the processor executes a "jump indirect" to memory location 47_8. The state of the Interrupt On flag is unaltered.

SECTION IV
INPUT/OUTPUT

INTRODUCTION

In order for the processor to perform useful work for the user, there must be some method for the program to transfer information outside the machine. The Input/Output (I/O) instruction set provides this facility. There are eight I/O instructions which allow the program to communicate with I/O devices, control the I/O interrupt system, control certain processor options, and to perform certain processor functions.

The microNOVA computers have a 6-bit device selection network, corresponding to bits 10-15 in the I/O instruction format. Each device is connected to this network in such a way that the device will only respond to commands with its own device code. Each device also has two flags, Busy and Done, which control its operation. When the Busy and Done flags are both zero, the device is idle and cannot perform any operations. To start a device, the program must set the Busy flag to 1 and the Done flag to 0. When a device has finished its operation, it sets its Busy flag to 0 and its Done flag to 1. The format for the I/O instructions is illustrated below.

0	1	1	AC	OP CODE	CONTROL	DEVICE CODE
0	1	2	3 4	5 6 7	8 9	10 11 12 13 14 15

Bits 0-2 are 011, bits 3-4 specify the AC, bits 5-7 contain the operation code, bits 8-9 control the Busy and Done flags in the device, and bits 10-15 specify the code of the device. The six bits provided for the device code in the I/O format mean that 64 unique device codes are available for use. Some of these device codes, however, are reserved for the CPU and certain processor options. The remaining device codes are available for referencing I/O units. Some of the codes have been assigned to specific devices by Data General and the assembler recognizes mnemonics for these devices. A complete listing of device codes, the devices assigned to these codes, and the mnemonics assigned to the devices is contained in Appendix A.

OPERATION OF I/O DEVICES

In general, the operation of all I/O devices is done by manipulating the Busy and Done flags. In order to operate a device, the program must first ensure that the device is not currently performing some operation. After the program has determined that the device is available, it can start an operation on the device by setting the Busy flag to 1 and the Done flag to 0. Once a device has completed its operation, and set its Busy flag to 0 and its Done flag to 1, it is available for another operation. The program can determine this condition in one of two ways. By using the I/O SKIP instruction, the program can test the status of the Busy and Done flags. Another way is to utilize the interrupt system that is standard on the microNOVA computers. The interrupt system is made up of an interrupt request line to which each I/O device is connected, an Interrupt On flag in the CPU, and a 16-bit interrupt priority mask.

The Interrupt On flag controls the status of the interrupt system. If the flag is set to 1, the CPU will respond to and process interrupts. If the flag is set to 0, the CPU will not respond to any interrupts.

Interrupt requests can be initiated in three ways. A stack interrupt request is initiated if a push operation places data into a word whose address is an integer multiple of 256. A real-time clock interrupt request is initiated every 2.4 milliseconds if the real-time clock is enabled. An I/O interrupt request is initiated by an I/O device when it completes its operation. Upon completing the operation, the device sets its Busy flag to 0 and its Done flag to 1. At this time, the device also places an interrupt request on the interrupt request line, provided that the bit in the interrupt priority mask which corresponds to the priority level of the device is 0. If the mask bit is 1, the device sets its Busy flag to 0 and its Done flag to 1, but does not place an interrupt request on the interrupt request line.

If the Interrupt On flag is 1 at the time the processor completes execution of any instruction, the processor honors any requests on the interrupt request line. If the Interrupt On flag is 0, the CPU does not look at the interrupt request line; it just goes on to the next sequential instruction. The CPU honors an interrupt request by setting the Interrupt On flag to 0 so that no interrupts can interrupt the first part of the interrupt service routine. The CPU then places the updated program counter in memory location 0 and executes a "jump indirect" to memory location 3, 2, or 1, depending upon the type of interrupt request. Stack interrupt requests cause a "jump indirect" to memory location 3. Real-time clock interrupt requests cause a "jump indirect" to memory location 2. I/O interrupt requests cause a "jump indirect" to memory location 1. It is assumed that these memory locations contain the address, either direct or indirect, of the correct interrupt service routine.

If more than one type of interrupt request occurs at the same time, the priority is: stack interrupt first, real-time clock interrupt second, and I/O interrupt third.

Once the CPU has transferred control to the interrupt service routine, it is up to that routine to save any accumulators that will be used, save the carry bit if it will be used, determine which device requested the interrupt (if it was an I/O interrupt request), and then service the interrupt. The determination of which device needs service can be done by I/O SKIP instructions or the routine can use the INTERRUPT ACKNOWLEDGE instruction.

If more than one device is requesting service, the code returned by an INTERRUPT ACKNOWLEDGE instruction is the code of that device requesting an interrupt which is physically closest to the CPU on the I/O bus.

After servicing the device, the interrupt routine should restore all saved values, set the Interrupt On flag to 1, and return to the interrupted program. The instruction that sets the Interrupt On flag to 1 (INTERRUPT ENABLE) allows the processor to execute one more instruction (if the INTERRUPT ENABLE instruction changed the condition of the Interrupt On flag) before the next interrupt can take place. In order to prevent the interrupt service routine from going into a loop, this next instruction should be the instruction that returns control to the interrupted program.

Since the updated value of the program counter was placed in location 0 by the CPU upon honoring the interrupt, all the interrupt routine has to do, after restoring the AC's and the carry bit, is execute an INTERRUPT ENABLE instruction and a "jump indirect" to location 0 and control will be returned to the interrupted program.

PRIORITY INTERRUPTS

If the Interrupt On flag remains 0 throughout the interrupt service routine, the interrupt routine cannot be interrupted and there is only one level of device priority. This level is determined by either the order in which the I/O SKIP instructions are issued or (if the INTERRUPT ACKNOWLEDGE instruction is used) by the physical location of the devices on the bus. In a system with devices of widely differing speed, such as a teletypewriter versus a diskette, the programmer may wish to set up a multiple level interrupt scheme. Hardware and instructions are available on the microNOVA computers to allow the implementation of up to sixteen levels of priority interrupts.

Each of the I/O devices is connected to a bit in the 16-bit priority mask. Devices which operate at roughly the same speed are connected to the same bit in the mask. Even though the standard mask bit assignments have the higher numbered bits assigned to lower speed devices, no implicit priority ordering is intended. The manner in which these priority levels are ordered is completely up to the programmer. The listing of device codes in Appendix A also contains the standard Data General mask bit assignments.

The condition of the priority mask is altered by the MASK OUT instruction. If a bit in the priority mask is set to 1, then all devices in the priority level corresponding to that bit will be prevented from requesting an interrupt when they complete an operation. In addition, all pending interrupt requests from devices in that priority level are disabled.

To implement a multiple priority level interrupt handler, the interrupt handler must be written in such a way that it may be interrupted without damage. For this to be possible, the main interrupt routine must save return information upon receiving control. The return information consists of the four accumulators, the carry bit, and the return address. This information should be stored in a unique place each time the interrupt handler is entered so that one level of interrupt does not overlay the return information that belongs to a lower priority level. The stack facility of the microNOVA computers enables this return information to be convienently stored in a standard form. After saving the return information, the interrupt routine must determine which device requires service and transfer control to the correct service routine. This can be done in the same manner as for a single level interrupt handler.

After the correct service routine has received control, that routine should save the current priority mask, establish the new priority mask, and enable the interrupt system with the INTERRUPT ENABLE instruction. After servicing the interrupt, the routine should disable the interrupt system with the INTERRUPT DISABLE instruction, reset the priority mask, restore the accumulators and the carry bit, enable the interrupt system, and return control to the interrupted program.

DATA CHANNEL

Handling data transfers between external devices and memory under program control requires the execution of several instructions for each word transferred. To allow greater data transfer rates, the microNOVA computers contain a data channel through which a device, at its own request, can gain direct access to memory using a minimum of processor time. At the maximum input rate of one word every 6.7 microseconds or 149,254 words per second, and at the maximum output rate of one word every 5.8 microseconds or approximately 172,414 words per second, the data channel effectively stops the processor, but at lower rates, processing continues while data is being transferred.

When a device is ready to send or receive data, it requests access time via the channel. At the beginning of every memory cycle, the processor synchronizes any requests that are then being made. At certain specified points during the execution of an instruction, the CPU pauses to honor all previously synchronized requests. When a request is honored, a word is transferred directly via the channel from the device to memory or from memory to the device without specific action by the program. All requests are honored according to the relative position of the requesting devices on the I/O bus. That device requesting data channel service which is physically closest on the bus is serviced first, then the next closest device, and so on, until all requests have been honored. The synchronization of new requests occurs concurrently with the honoring of other requests. If a device continually requests the data channel, that device can prevent all devices further out on the bus from gaining access to the channel.

Following completion of an instruction, the processor handles all data channel requests, and then honors all outstanding I/O interrupt requests. After all data channel and I/O interrupt requests have been serviced, the processor continues with the next sequential instruction.

ADDRESSING NONEXISTENT DEVICES

The six-line device selection network of the microNOVA computers can address 64 separate device codes. It is possible, however, that some of these device codes will not have devices associated with them. If an attempt is made to issue an input I/O instruction to a nonexistent device, the CPU functions as if the device exists and has all the bits of its A, B, and C input buffers set to 1. If an attempt is made to issue an output I/O instruction to a nonexistent device, the CPU functions as if the device exists and no indication is given that it does not exist. If an attempt is made to test the status of the Busy flag or the Done flag of a nonexistent device, the CPU functions as if the device exists and has both its Busy flag and its Done flag set to 1.

CODING AIDS

The set of I/O instructions has options that can be obtained by appending mnemonics to the standard mnemonics. These optional mnemonics and their result are given below.

CLASS ABBREV.	CODED CHARACTER	RESULT BITS	OPERATION
f	(option omitted)	00	Does not affect the Busy and Done flags
	S	01	Start the device by setting Busy to 1 and Done to 0
	C	10	Idle the device by setting both Busy and Done to 0
	P	11	Pulse the special in-out bus control line - The effect, if any depends on the device

The I/O SKIP instruction enables the programmer to make decisions based upon the values of the Busy and Done flags. Which test is performed is based upon the value of bits 8-9 in the instruction. Bits 8-9 can be set by appending an optional mnemonic to the I/O SKIP mnemonic. The optional mnemonics and their results are given below.

CLASS ABBREV.	CODED CHARACTER	RESULT BITS	OPERATION
t	BN	00	Tests for Busy = 1
	BZ	01	Tests for Busy = 0
	DN	10	Tests for Done = 1
	DZ	11	Tests for Done = 0

I/O INSTRUCTIONS

No I/O Transfer

NIO *[f]* *device*

0	1	1	0 0	0	0	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

The Busy and Done flags in the specified device are set according to the function specified by F.

Data In A

DIA *[f]* *ac,device*

0	1	1	AC	0	0	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

The contents of the A input buffer in the specified device are placed in the specified AC. After the data transfer, the Busy and Done flags are set according to the function specified by F.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the AC that do not receive data are set to 1.

Data In B

DIB *[f]* *ac,device*

0	1	1	AC	0	1	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

The contents of the B input buffer in the specified device are placed in the specified AC. After the data transfer, the Busy and Done flags are set according to the function specified by F.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the AC that do not receive data are set to 1.

Data In C

DIC *[f]* *ac,device*

0	1	1	AC	1	0	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

The contents of the C input buffer in the specified device are placed in the specified AC. After the data transfer, the Busy and Done flags are set according to the function specified by F.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the AC that do not receive data are set to 1.

Data Out A

DOA *[f]* *ac,device*

0	1	1	AC	0	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

The contents of the specified AC are placed in the A output buffer of the specified device. After the data transfer, the Busy and Done flags are set according to the function specified by F. The contents of the specified AC remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

Data Out B

DOB *[f]* *ac,device*

0	1	1	AC	1	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

The contents of the specified AC are placed in the B output buffer of the specified device. After the data transfer, the Busy and Done flags are set according to the function specified by F. The contents of the specified AC remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

Data Out C

DOC *[f]* *ac,device*

0	1	1	AC	1	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

The contents of the specified AC are placed in the C output buffer of the specified device. After the data transfer, the Busy and Done flags are set according to the function specified by F. The contents of the specified AC remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

I/O Skip

SKP *[t]* *device*

0	1	1	0 0	1	1	1	T	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15

If the test condition specified by T is true, the next sequential word is skipped.

CENTRAL PROCESSOR FUNCTIONS

I/O instructions with a device code of 77_8 perform a number of special functions rather than controlling a specific device. In all but the I/O SKIP instruction, I/O instructions with a device code of 77_8 use bits 8-9 to control the condition of the Interrupt On flag. An I/O SKIP instruction with a device code of 77_8 uses bits 8-9 to test the state of the Interrupt On flag. The mnemonics are the same as for normal I/O instructions. The table below gives the result of these bits for instructions with a device code of 77_8.

CLASS ABBREV.	CODED CHARACTER	RESULT BITS	OPERATION
f	(option omitted)	00	Does not affect the state of the Interrupt On flag
	S	01	Set the Interrupt On flag to 1
	C	10	Set the Interrupt On flag to 0
	P	11	Does not affect the state of the Interrupt On flag
t	BN	00	Tests for Interrupt On = 1
	BZ	01	Tests for Interrupt On = 0
	DN	10	Never skip
	DZ	11	Always skip

The device code of 77_8 deals mainly with processor functions and has, therefore, been given the mnemonic CPU. In addition, many of the I/O instructions that reference this device code have been given special mnemonics. While these special mnemonics are functionally equivalent to the corresponding I/O instructions with a device code of 77_8, there is the following limitation; the mnemonics for controlling the state of the Interrupt On flag cannot be appended to them.

If the programmer wishes to alter the state of the Interrupt On flag while performing a MASK OUT instruction, for example, he must issue the appropriate I/O instruction - **DOB** *[f] ac*, **CPU** - instead of the corresponding special mnemonic **MSKO** *ac*. If the special mnemonic is used, bits 8-9 are set to 00. In describing the instructions, the special mnemonic for the corresponding I/O instruction will be given first, followed by the I/O instruction.

Interrupt Enable

INTEN

NIOS CPU

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

The Interrupt On flag is set to 1.

If the state of the Interrupt On flag is changed by this instruction, the CPU allows one more instruction to execute before the first I/O interrupt can occur.

Interrupt Disable

INTDS

NIOC CPU

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

The Interrupt On flag is set to 0.

Interrupt Acknowledge

INTA *ac*

DIB *[f]* *ac*, **CPU**

0	1	1	AC		0	1	1	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The six-bit device code of that device requesting an interupt which is physically closest to the CPU on the I/O bus is placed in bits 10-15 of the specified AC. Bits 0-9 of the specified AC are set to 1. If no device is requesting an interrupt, bits 0-15 of the specified AC are set to 1. After the transfer, the Interrupt On flag is set according to the function specified by F.

Mask Out

MSKO *ac*

DOB *[f]* *ac*, **CPU**

0	1	1	AC		1	0	0	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The contents of the specified AC are placed in the priority mask. After the transfer, the Interrupt On flag is set according to the function specified by F. The contents of the specified AC remain unchanged.

NOTE *A 1 in any bit disables interrupt requests from devices in the corresponding priority level.*

Changes in the priority mask do not take effect until after the instruction following the MASK OUT.

Reset

IORST

DOA *[f]* 0,CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	0	1	0	F		1	1	1	1	1	1

The Busy and Done flags in all I/O devices are set to 0. The 16-bit priority mask is set to 0. The real-time clock is disabled. All interfaces based on the mN603 IOC circuit have their device codes and priority mask bits initialized. The Interrupt On flag is set according to the function specified by F.

NOTES *The assembler recognizes the mnemonic IORST as equivalent to the instruction* **DOAC 0,CPU**.

If the mnemonic DOA is used to perform this function, AC0 must be specified as the accumulator. Regardless of how it is coded, during execution of the instruction, the contents of AC0 remain unchanged.

Halt

HALT

DOC 0,CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		1	1	0	F		1	1	1	1	1	1

The Interrupt On flag is set to 1 and the processor is stopped. While in the stopped state, the CPU will honor data channel requests. In addition, because this instruction sets the interrupt on flag to 1, the CPU will honor program interrupt requests while in the stopped state.

NOTES *The optional mnemonics S, C, and P have no effect when appended to a* **DOC 0,CPU** *instruction.*

If the mnemonic DOC is used to perform this function, AC0 must be specified as the accumulator. Regardless of how it is coded, during execution of the instruction, the contents of AC0 remain unchanged.

The actions of a microNOVA computer system after execution of a HALT instruction depend upon whether or not either the hand-held console or the console debug option are installed in the system. If either of these devices are present, then immediately after the HALT instruction is executed, control will be transferred to the appropriate console software. Otherwise, the CPU will remain in the stopped state, waiting for an interrupt.

CPU Skip

SKP *[t]* CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	1	1	1	T		1	1	1	1	1	1

If the test condition specified by T is true, the next sequential word is skipped.

REAL-TIME CLOCK

The real-time clock (RTC) feature of the microNOVA computers generates periodic interrupts when enabled.

When enabled, a real-time clock interrupt request is initiated every 2.4 milliseconds. Upon receiving a real-time clock interrupt request, the CPU sets the Interrupt On flag to 0, places the updated program counter in memory location 0, and executes a "jump indirect" to memory location 2.

The real-time clock is enabled and disabled with two instructions to device code 77_8. When the clock is first enabled, the first interrupt request can occur at any time within the first 2.4 milliseconds after the enabling instruction. After that, the clock will generate an interrupt request every 2.4 milliseconds until it is disabled.

Real-time Clock Enable

RCTEN

DOA *[f]* 2,CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	0	0	1	0	F		1	1	1	1	1	1

The real-time clock is enabled.

After the clock is enabled, the Interrupt On flag is set according to the function specified by F.

Real-time Clock Disable

RTCDS

DOA *[f]* 1,CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	1	0	1	0	F		1	1	1	1	1	1

The real-time clock is disabled.

After the clock is disabled, the Interrupt On flag is set according to the function specified by F.

POWER FAIL/AUTO-RESTART

In the microNOVA computers, when power fails and is later restored, the state of the memory, the accumulators, the program counter, and the various flags in the CPU is preserved if the battery backup option is present. The power fail/auto-restart circuitry provides a "fail-soft" capability in the event of unexpected power loss.

In the event of power failure, there is a delay of one to two milliseconds before the processor shuts down. The power fail circuitry senses the imminent loss of power and requests an I/O interrupt. The power fail circuitry has no device code, so when the I/O interrupt handler issues an INTERRUPT ACKNOWLEDGE instruction, the specified accumulator will have all its bits set to 1. The I/O interrupt handler should then transfer control to the power fail routine.

The power fail routine can then use the delay to set up the return linkages needed to restart the interrupted program. The power fail routine should then execute a HALT instruction. This HALT instruction will clear the power fail interrupt and leave the system in the stopped state until power is restored. One to two milliseconds is enough time to execute 500 to 1000 instructions on the microNOVA computers, so there is more than enough time to perform the power fail routine.

When power is restored, the action taken by the automatic restart circuitry depends primarily upon the position of the power switch on the front panel. If the switch is in the "run" position when power is restored, the CPU acts as if it had just executed a HALT instruction.

If the switch is in the "lock" position when power is restored, the action taken depends primarily upon the state of the battery backup option.

If the batteries have run down during the power failure, or if the battery backup option is not present, then, when power is restored, the CPU acts as if it had just executed a HALT instruction.

If the batteries have not run down during the power failure, the action depends upon the state of jumpers on the CPU board. These jumpers offer several alternatives for the automatic restart procedure.

The CPU can be directed to retrieve the contents of memory location 077777_8, set bit 15 of the retrieved contents to 0, and then use the result as an intermediate address in the effective address calculation. After determining the effective address, the CPU performs a "jump" to that location. The memory locations 077776_8 and 077777_8, can be contained in random-access memory (RAM), read-only memory (ROM), or set with jumpers on the CPU board.

Alternatively, the CPU can be directed to perform a program load sequence. The address of the device to be used and the type of program load to be performed can be contained in ROM or set with jumpers on the CPU board.

For a further description of the auto-restart alternatives and the jumper settings which enable them, consult the Technical Reference for microNOVA Computer Systems *(DGC no. 014-000073)*.

HAND-HELD CONSOLE

The hand-held console available with the microNOVA computers is an I/O device that, when used in conjunction with the standard software, allows the user to control and monitor the actions of the computer system. However, because the hand-held console is an I/O device, and not a direct extension of the CPU, it may be programmed by the user to augment the standard software or to totally redefine its actions.

The hand-held console is a device that looks like a small calculator. It has 20 keys and a 6-digit, 7-segment LED display. The controller for the hand-held console occupies one slot in the computer chassis. This controller will initiate an I/O interrupt request each time a key on the hand-held console is struck. The console is connected to the controller by a 16-conductor ribbon cable.

The hand-held console is driven by a controller that contains 256 locations of read-only memory (ROM) that respond to addresses 077400-077777$_8$. If the microNOVA computer system contains read-write memory (also called random-access memory or RAM) at those locations, the console controller ROM takes precedence and the RAM is disabled. Within the 256 locations of ROM are 16 locations of RAM. This array of ROM/RAM is used by the standard console software to implement the console actions described in section V.

The standard console software has two entry points. Memory location 077777$_8$ contains the address of the location that will receive control upon system initialization or after an automatic restart. If the user wishes to pass interrupts from the hand-held console on to the standard console software, then he should transfer control to the location whose address is in memory location 077776$_8$ whenever a hand-held console interrupt is received.

One of the 16 locations of RAM is at memory address 077576$_8$. The contents of this location are divided into octal digits and continuously displayed in the 6 digits on the console. The left-hand digit corresponds to bit 0, and the right-hand digit corresponds to the octal digit made from bits 13-15. This memory location can be retrieved by a LOAD instruction that references memory location 077576$_8$. The display can be altered at any time by placing a new value in location 077576$_8$.

Another one of the RAM locations is at memory address 077577$_8$. This location is used to hold the console switches register. This location can be retrieved by an I/O instruction or by a LOAD instruction that references location 077577$_8$.

The controller also contains a 5-bit function register that contains the function code of the most recently struck console key.

Instructions

Three I/O instructions are used to program the hand-held console. One instruction is used to retrieve the current contents of the console switches register. One instruction is used to retrieve the function code of the most recently struck console key. The remaining instruction is used to light the decimal point to the right of the left-hand digit on the console.

The device code for the hand-held console is 4 and it has the mnemonic HHC. Its priority mask bit is bit 5.

The device flag commands control the hand-held console's Busy and Done flags in the following manner:

f=S	Set the Busy flag to 1 and the Done flag to 0.
f=C	Set both the Busy and Done flags to 0.
f=P	No effect.

Read Switches

READS *ac*

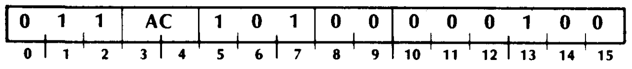
DIA *[f]* *ac,* **HHC**

0	1	1	AC		0	0	1	0	0	0	0	0	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

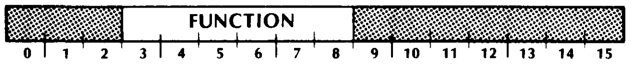
The current value of the console switches register is retrieved from memory location 077577$_8$ and placed in the specified AC. After the transfer, the function specified by F is performed.

Read Function

DIC *[f]* *ac*, **HHC**



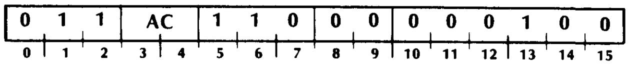
The function code of the most recently struck console key is placed in bits 3-7 of the specified AC. Bits 0-2 are set to 0. Bits 8-15 are unpredictable. After the transfer, the function specified by F is performed. The format of the specified AC is as follows:



Bits	Name	Contents
0-2	----	Reserved for future use.
3-7	Function	Function code of the most recently struck console key.
	00 000	0/AC0
	00 001	1/AC1
	00 010	2/AC2
	00 011	3/AC3
	00 100	4/FP
	00 101	5/SP
	00 110	6/SWITCHES
	00 111	7/ADDR
	01 000	LAST
	01 001	NEXT
	01 010	MEM
	01 011	CLR D
	01 100	START
	01 101	STOP
	01 110	CONT
	01 111	DEP
	10 000	RESET
	10 001	•
	10 010	/
	10 011	PR LOAD
8-15	----	Reserved for future use.

Light Decimal Point

DOC *[f]* *ac*, **HHC**



The decimal point to the right of the right-hand digit on the console is lit. The contents of the specified AC are ignored and remain unchanged. After the light is lit, the function specified by F is performed.

NOTE *In order to keep the decimal point visible, this instruction must be issued at least once every 16 milliseconds.*

Programming

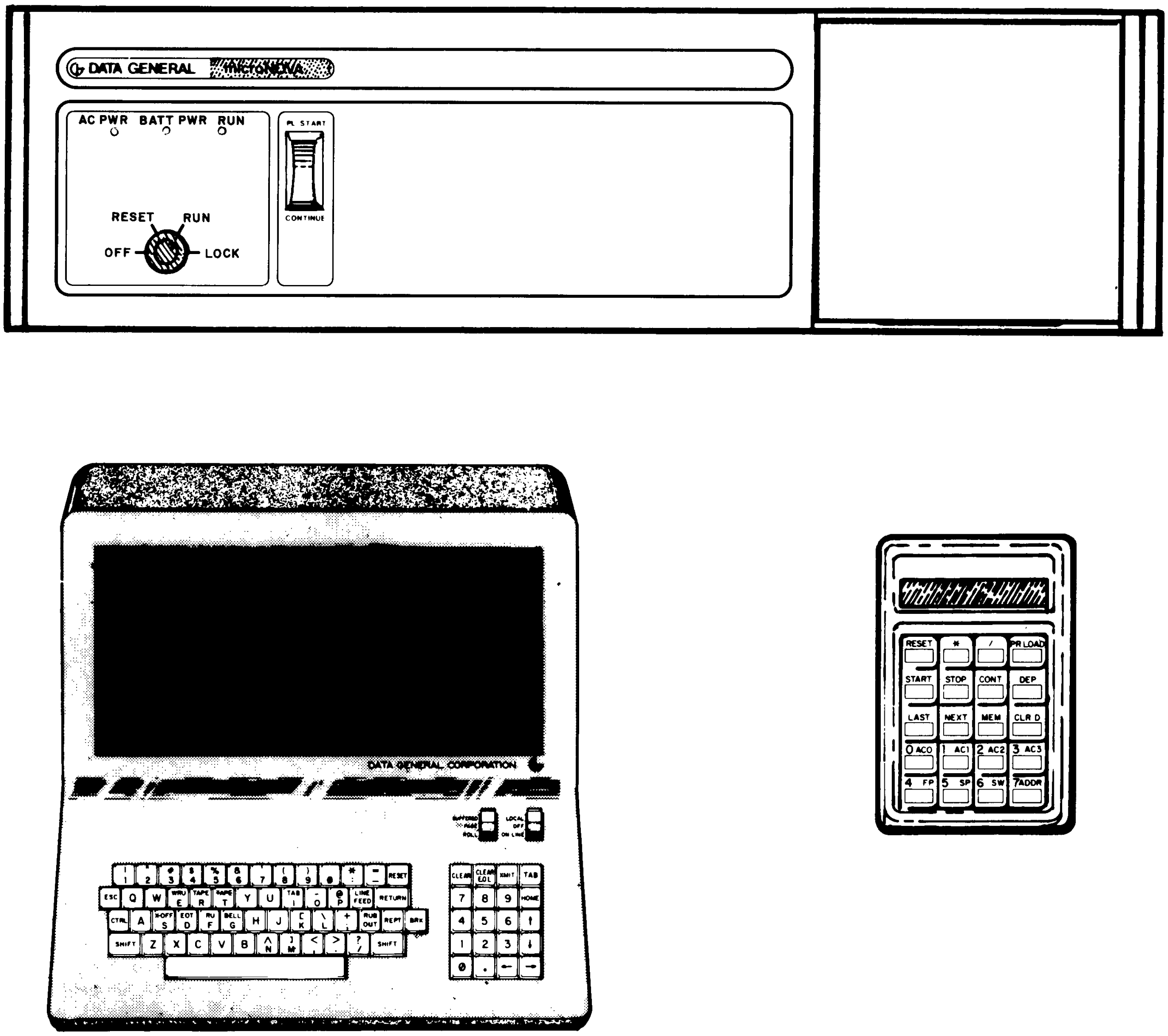
Each time a key is struck on the hand-held console, the function code for that key is placed in the function code register, the controller's Done flag is set to 1, and an I/O interrupt request is initiated.

The function code can then be read by issuing a READ FUNCTION instruction (DIC). The Done flag should be set to 0 with either a Start or Clear command. This allows the next key strike to initiate another I/O interrupt request.

Considerations

The RESET key on the hand-held console is the only key that performs its action without software intervention. If the front panel power switch is in the RUN position when this key is struck, program execution is stopped at the end of the current instruction and the system is initialized as if the front panel power switch were turned to the RESET position. An I/O RESET instruction is executed. The CPU is stopped. Control is then transferred to the standard hand-held console software.

If the front panel power switch is in the LOCK position when the RESET key is struck, the function code for the RESET key is placed in the function register and an I/O interrupt request is initiated.


DATA GENERAL microNOVA
AC PWR
BATT PWR
RUN
RESET
RUN
OFF
LOCK
PL START
CONTINUE
DATA GENERAL CORPORATION
RESET
*
/
PR LOAD
START
STOP
CONT
DEP
LAST
NEXT
MEM
CLR D
0 AC0
1 AC1
2 AC2
3 AC3
4 FP
5 SP
6 SW
7 ADDR

DG-02492

SECTION V
OPERATION

A microNOVA computer system can be controlled at three levels: front panel or CPU board controls, hand-held console, and system console. The front panel and the CPU board controls allow the system to be operated at the basic level of controlling the power and initiating a program load sequence. The hand-held console allows memory locations and the internal registers of the CPU to be examined and modified. It also allows the system to be started and stopped. The system console, whether it be a teletypewriter or a CRT terminal, can, if the controller is equipped with the console debug option, do all that the hand-held console can do plus allow the setting of breakpoints in the user's program so that he can follow the process of the program step-by-step.

FRONT PANEL

The front panel of the microNOVA computer chassis contains a 4-position, locking, power switch; a 2-position rocker switch; and 3 indicator lights.

Power Switch

The power switch has 4 postions: OFF, RESET, RUN, and LOCK. With the switch in the OFF position, all power to the CPU is off and the system will not run. Turning the switch to the RESET position initializes the CPU. As long as the switch is in the RESET position, power is being supplied to the system, but the CPU is held in the HALT state. Turning the switch to the RUN postion allows the CPU to leave the HALT state and begin operating. Turning the switch to the LOCK position allows the key to be removed. The system will automatically restart after a power failure only if the switch is in the LOCK position.

Rocker Switch

The rocker switch has 2 positions: PL/START, and CONTINUE. The switch is spring loaded and will return to the center, neutral position after it is released. Pressing the switch to the PL/START position either initiates a program load sequence or causes the system to start at a predetermined memory location. The action taken depends upon the state of jumpers on the CPU board. For a description of the jumper settings, consult the Technical Reference for microNOVA Computer Systems *(DGC no. 014-000073)*. Pressing the switch to the CONTINUE position causes the system to continue program execution at the location addressed by the current contents of the program counter.

Indicator Lights

The 3 indicator lights are labelled AC POWER, BATT POWER, and RUN. When lit, the AC POWER light means that ac power is being supplied to the system. When lit, the BATT POWER light means that the ac power has failed and that power from the battery backup option is being used to refresh the memories. When lit, the RUN light means that the CPU is executing either instructions or data channel cycles.

CPU BOARD CONTROLS

If a microNOVA CPU board has been purchased for operation in an environment that does not include the standard front panel, it is supplied with on-board controls that are analogous to those found on the front panel.

These controls consist of a slide switch, 3 push-button switches, and 3 LED indicators. The slide switch simulates the LOCK position of the power switch. The 3 push-buttons implement the RESET, PL/START, and CONTINUE functions. The 3 LED indicators are the same as the 3 lights on the front panel.

HAND-HELD CONSOLE

The hand-held console is a device that looks like a small calculator. It has 20 keys and a 6-digit, 7-segment LED display. It is connected to the computer chassis by a 16-conductor ribbon cable. It can be used when attached to the front panel of the chassis or when held in the user's hand. When used with the standard software, the hand-held console allows the user to reset, start, and stop the system, and to examine and modify memory locations or the internal registers of the CPU. The actions of the hand-held console when used with the standard software are described below.

Display

The 6-digit display displays a 16-bit number as 6 octal digits. The left-hand digit corresponds to bit 0, the next digit corresponds to bits 1-3, the next digit corresponds to bits 4-6, and so on. The decimal point to the right of the right-hand digit will light if the console software received control either as result of the CPU executing a HALT instruction or the console STOP key being struck.

Key Pad

The 20-key key pad of the hand-held console is divided into 9 function keys and 11 location keys. The function keys control such functions as RESET, START, STOP, and CONTINUE. The location keys can either define the location of an EXAMINE or DEPOSIT function, or can be used to enter a number into the display.

Function Keys

When the CPU is running, all the keys except PR LOAD and START are enabled. When the CPU is stopped (indicated by a lighted decimal point to the right of the right-hand digit), all the keys are enabled.

RESET

If the front panel power switch is in the RUN position when this key is struck, program execution is stopped at the end of the current instruction and the system is initialized as if the front panel switch were turned to the RESET position. An I/O RESET instruction is executed and the CPU is stopped. Control is then transferred to the hand-held console software.

If the front panel power switch is in the LOCK position when this key is struck, program execution is stopped at the end of the current instruction and the CPU is placed in the stopped state.

*

This key is reserved for future use.

/

This key is reserved for future use.

PR LOAD

When this key is struck, the bootstrap loader is placed in memory locations 2-37_8. After an I/O RESET instruction is executed, execution of the bootstrap loader is started at location 2. The device code of the program load device is taken from the 2 right-hand digits of the display. The mode of the program load is taken from the left-hand digit of the display. If this digit is 0, the load is performed via programmed I/O. If this digit is 1, the load is performed via the data channel.

START

When this key is struck, the current contents of the display are placed in the program counter, an I/O RESET instruction is executed and program execution is started with the instruction addressed by the updated value of the program counter.

STOP

When this key is struck, program execution is stopped at the end of the current instruction and the Interrupt On flag is set to 0. The display shows the value of the carry bit in the left-hand digit and the address of the last instruction executed in the right-hand 5 digits.

CONT

When this key is struck, program execution is continued by executing a **JMP @ 0** instruction. An INTERRUPT ENABLE instruction is executed before the JUMP.

NOTE *The hand-held console is an I/O device and the only way that the console software can receive control is via an interrupt. Since interrupts always place the updated program counter in location 0, a* **JMP @ 0** *instruction is all that is needed to continue program execution.*

DEP

After this key is struck, if the next key struck is a location key, the current contents of the display will be placed in that location. If the next key struck is a function key, that function will be performed and this strike of the DEP key will have no effect.

CLR D

The display is set to 0 and the numeric entry mode is entered. The bottom two rows of keys will function as an 8-key, octal key-pad. For each key struck, the display is shifted left one digit and the struck digit is placed in the right-hand digit. After 6 digits have been entered, the software will refuse further digits unless the CLR D key is struck again.

Location Keys

For all location keys, if the DEP key was struck immediately previous, the location key defines the destination of a deposit and the contents of the display will be placed in that location. Otherwise, the contents of the location will be retrieved and placed in the display.

ADDR

The location defined is the console memory address register. This register contains the address that will be used with the MEM, NEXT, and LAST keys.

SWITCHES

The location defined is the console switches register. This register is accessible to the user program via the READ SWITCHES instruction.

SP

The location defined is the stack pointer.

FP

The location defined is the frame pointer.

AC3

The location defined is accumulator 3.

AC2

The location defined is accumulator 2.

AC1

The location defined is accumulator 1.

AC0

The location defined is accumulator 0.

MEM

If the DEP key was struck immediately previous, the location defined is the memory location addressed by the console memory address register. Otherwise, the contents of the display are placed in the console memory address register and the location defined is the memory location addressed by the updated value of the console memory address register.

NEXT

The contents of the console memory address register are increased by 1 and the location defined is the memory location addressed by the updated value of the console memory address register.

LAST

The contents of the console memory address register are decreased by 1 and the location defined is the memory location addressed by the updated value of the console memory address register.

CONSOLE DEBUG OPTION

The console debug option is an option available with the asynchronous interface board that allows the use of the system console to monitor program execution and to examine and modify memory locations and CPU internal registers.

The console debug option consists of a program contained in 256 locations of read-only memory (ROM) that respond to addresses 077400-077777_8. If the microNOVA computer system contains read-write memory (also called random-access memory or RAM) at those locations, the console debug option ROM takes precedence and the RAM is disabled. The console debug option ROM has the same addresses as the ROM for the hand-held console. For this reason both consoles may not be present in a system at the same time.

The console debug option receives control when any of the following events occur: the front panel power switch is turned to the RESET position and then to either the RUN position or the LOCK position; the CPU executes a HALT instruction; or a breakpoint is encountered. Additionally, if the front panel power switch is in the LOCK position and power fails, the automatic restart feature may be directed to transfer control to the console debug option when power is restored.

The console debug option indicates to the user that it has control of the system by typing the program counter of the interrupted program in octal, followed by a carriage return, a line feed, and a ! character. At this point the user may examine and modify any memory location, any of the four accumulators, the stack pointer, the frame pointer, and certain status indicators.

In addition, the user may define a point in his program called a breakpoint. After the user directs the console debug option to start or restart his program, the program will be executed until it is stopped or the breakpoint is reached. If the breakpoint is reached, control is transferred to the console debug option and the breakpoint is cleared. The user's instruction at the breakpoint location is not executed.

The locations that can be examined and modified by the user are called cells. These cells are of two types: internal CPU registers, and memory locations. The action of examining a cell is referred to as "opening" the cell. The action of releasing a cell is referred to as "closing" the cell. After a cell has been opened, it may be modified and then closed, or closed without modification. Only one cell may be open at any time. After the command has been given to open a cell, its contents are typed out in octal.

Opening Internal Cells

The command to open an internal CPU cell is of the form *n* **A**, where *n* is an octal integer in the range 0-17_8. The different values of *n*, together with which internal cell they open, are tabulated below.

Octal Integer	Internal Cell
0	Accumulator 0
1	Accumulator 1
2	Accumulator 2
3	Accumulator 3
4	The program counter of the interrupted program.
5	Stack Pointer
6	Frame Pointer
7	CPU and console controller (TTO) status where: Bits 0-12 are reserved for future use. Bit 13 is status of the carry bit when the console debug option received control. Bit 14 is status of Interrupt On flag when the console debug option received control Bit 15 is status of TTO Done flag when the console debug option received control
10	Address of a location in the first 256 words of main memory that can be used by the console debug option for breakpoint transfers.
11	Address of the most recent breakpoint
12	User instruction at the address of the most recent breakpoint
17	Contents of memory location 077577_8.

Opening Memory Cells

The command to open a memory cell is of the form *addr* /, where *addr* is an octal number which is the address of the desired memory location or an arithmetic expression made up of octal numbers separated by plus and minus signs. Leading zeros need not be typed. The expression is evaluated and the low-order 15 bits of the result are used as the address of the desired memory location. The period character (.) can be used to signify the address of the most recently opened memory location.

Once a cell has been opened, the contents of that cell may be used to specify the location of the next memory location to be opened. If a plus or minus sign is typed followed by an octal number or an arithmetic expression, followed by a slash (/), the octal number or the result of the arithmetic expression is added to or subtracted from the contents of the open cell and this result is used to address the new cell. The current open cell is closed without modification. The new cell is opened and its contents are typed out in octal. If only a slash is typed, the current cell is closed without modification and the contents of that cell are used to address a new cell. The new cell is opened and its contents are typed out in octal.

Modifying a Cell

Once a cell has been opened, the user may modify the cell by typing the new value the cell is to contain, followed by a carriage return or a line feed. If a carriage return is typed, the cell will be closed and the console debug option will await the next command. If a line feed is typed, the cell will be closed and, if the cell was a memory location, the next higher memory location will be opened. If the cell was an internal register, it will be closed and the next higher numbered internal cell will be opened. If no new value is typed preceding the carriage return or line feed, the cell is closed without modification.

The new value may be specified by typing an octal number or an arithmetic expression made up of octal numbers separated by plus and minus signs. Leading zeros need not be typed. The expression will be evaluated and the low-order 16 bits of the result become the new value. The period character can be used to signify the address of the most recently opened memory location.

If a plus or minus sign is typed as the first character of the new value, then the value of the typed expression is added to or subtracted from the contents of the cell and the low-order 16 bits of this result become the new value.

Other Commands

There are five other commands to the console debug option that allow the user to set a breakpoint, clear a breakpoint, start the execution of a program, continue the execution of a program, and perform a program load.

Set Breakpoint

The command to set a program breakpoint is of the form *addr* **B**, where *addr* can be specified in the same manner as the address used to open a memory location.

NOTES *The console debug option will place a JUMP instruction, with the indirect bit set, in the memory location specified by addr. The location referenced by the indirect reference will be the location whose address is contained in CPU internal cell 10_8. If the user wishes to utilize the breakpoint capability, he must initialize CPU internal cell 10_8 to be the first 256 address of some unused word in the first 256 locations of main memory. Locations 0-3 should not be used for this purpose since they are used by the CPU's interrupt mechanism.*

Once a breakpoint has been set, it must be cleared before another breakpoint can be set. A breakpoint can be cleared either by encountering it during program execution or by using the **D** *command.*

Clear Breakpoint

The command to clear a breakpoint is of the form **D**. The contents of CPU internal cell 12_8 are placed in main memory at the address contained in CPU internal cell 11_8.

Start Execution

The command to start execution of a program is of the form *addr* **R**, where *addr* can be specified in the same manner as the address used to open a memory location. An I/O RESET instruction is executed and then control is transferred to the address specified by *addr*.

Continue Execution

The command to continue execution of the interrupted program is **P**. Control is transferred to the address contained in CPU internal cell 4.

Program Load

The command to perform a program load sequence is of the form *dev* **L**, where *dev* is the octal device code of the program load device.

Rubout

The RUBOUT or DEL key on the console keyboard may be used to correct commands as they are being typed. Typing this character effectively erases the right-most character in the command. Typing this character twice in succession effectively erases the two right-most characters in the command, and so on. Typing this character immediately after opening a cell has the same editing effect on the contents of the cell.

K

Typing the character **K** deletes the entire command. If a cell is open, it is closed without modification. The console debug option responds by typing a question mark (?).

PROGRAM LOADING

Before a program can be executed it must be brought into memory. This requires that a loading program already reside in memory. If the memory does not already contain a loading program, the operator can use the program load facility to bring in a "bootstrap loader" which can, in turn, read in a loading program.

There are two ways in which an operator can initiate a program load sequence. If the microNOVA computer system contains either the console debug option or a hand-held console, they can be used to initiate a program load sequence. If the system does not contain either of these devices, but the CPU board is equipped with the program load option, a program load sequence can be initiated with the PL/START switch.

Initiating a program load sequence in either of these ways deposits a 30_{10} word bootstrap loader into memory locations 2-37_8, places the device code of the program load device in accumulator 0, executes an I/O RESET instruction, and then begins sequential operation at memory location 2. This bootstrap loader will then read in data from the device whose device code is in AC0. The bootstrap loader can use either programmed I/O to read from a low-speed device or data channel transfers to read from a high-speed device.

To enter a loader program, the operator must first set up the device that is to be used and make its octal device code available to the program load facility being used. For the console debug option, this is done with the *dev* **L** command. For the hand-held console, this is done by entering the device code into the display. For the CPU board program load option, this is done by inserting jumpers on the CPU board.

The octal device code of the program load device should be placed in bits 10-15. The state of bit 0 indicates whether the device is a data channel device or a programmed I/O device. If the device is a data channel device, set bit 0 to 1. If the device is not a data channel device, set bit 0 to 0. After this is done, initiate the program load sequence. The bootstrap loader will be deposited into memory locations 2-37_8 and started at location 2.

If bit 0 is 1, the bootstrap loader starts the device for a data channel transfer by issuing an NIOS instruction and then loops at location 377_8 until a data channel transfer places a word into that location.

After a word has been placed in location 377_8, it is executed as an instruction. Typically, this word is either a HALT or a JUMP into the data that the data channel has placed in the first 377_8 memory locations.

NOTE *For proper program loading via the data channel, the device used must be initiated for reading into memory beginning at location 0 by an I/O RESET followed by an NIOS instruction. In addition, it is up to the device to stop reading after 256 words have been read.*

If bit 0 is a 0, the bootstrap loader reads the loader program via programmed I/O. The device must supply 8-bit data bytes, and each pair of bytes is stored as a single word in memory, wherein the first and second bytes read become the left and right halves of the word. To simplify the procedure, the bootstrap loader ignores leading null characters. It does not begin storing any words until it reads a non-zero synchronization byte. The first word following this synchronization byte must be the negative of the total number of words to be read, including the first word. The number of words to be read, including the first word may not be greater than 192_{10}. The bootstrap loader stores these words beginning at memory location 100_8. After storing the last word read, it transfers control to that location.

Listed below is the standard 30 word bootstrap loader for the microNOVA computers. This program is capable of loading in either of the manners described above.

The usual procedure is to use the bootstrap loader to bring in a larger program that sizes memory and then reads in the binary loader, storing it at the top of memory.

```
         LDA    1,C77     ;GET DEVICE MASK (000077)
         AND    0,1       ;ISOLATE DEVICE CODE
         COM    1,1       ;-DEVICE CODE - 1
LOOP:    ISZ    OP1       ;COUNT DEVICE CODE INTO ALL
         ISZ    OP2       ;I/O INSTRUCTIONS
         ISZ    OP3
         INC    1,1,SZR   ;DONE?
         JMP    LOOP      ;NO, INCREMENT AGAIN
         LDA    2,C377    ;YES, PUT JMP 377
                          ;INTO LOCATION 377
         STA    2,377
OP1:     060077           ;START DEVICE: (NIOS 0) - 1
         MOVL   0,0,SZC   ;LOW SPEED DEVICE?
                          ;(TEST SWITCH 0)
C377:    JMP    377       ;NO, GO TO 377
                          ;AND WAIT FOR CHANNEL
LOOP2:   JSR    GET+1     ;GET A FRAME
         MOVC   0,0,SNR   ;IS IT NON-ZERO?
         JMP    LOOP2     ;NO, IGNORE AND GET ANOTHER
LOOP4:   JSR    GET       ;YES, GET FULL WORD
         STA    1,  C377@;STORE STARTING AT 100 2'S
                          ;COMPLEMENT OF WORD
                          ;COUNT (AUTO-INCREMENT)
         ISZ    100       ;COUNT WORD - DONE?
         JMP    LOOP4     ;NO, GET ANOTHER
C77:     JMP    77        ;YES, - LOCATION COUNTER
                          ;AND JUMP
                          ;TO LAST WORD
GET:     SUBZ   1,1       ;CLEAR AC1, SET CARRY
OP2:
LOOP3:   063577           ;DONE?: (SKPDN 0) - 1
         JMP    LOOP3     ;NO, WAIT
OP3:     060477           ;YES, READ IN AC0: (DIAS 0,0) -1
         ADDCS  0,1,SNC   ;ADD 2 FRAMES SWAPPED -
                          ;GOT SECOND?
         JMP    LOOP3     ;NO, GO BACK AFTER IT
         MOVS   1,1       ;YES, SWAP THEM
         JMP    0,3       ;RETURN WITH FULL WORD
         0                ;PADDING
```

APPENDIX A
STANDARD I/O DEVICE CODES

OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME	OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
00				40			
01	----	----	Multiply/divide and stack	41			
02				42			
03				43			
04	HHC	5	Hand-held console	44			
05	PROG	9	PROM programmer	45			
06				46			
07				47			
10	TTI	14	TTY input	50	TTI1	14	Second TTY input
11	TTO	15	TTY output	51	TTO1	15	Second TTY output
12				52			
13				53			
14				54			
15				55			
16				56			
17				57			
20				60			
21				61			
22				62			
23				63			
24				64			
25				65			
26				66			
27				67			
30	DKT2	10	Third diskette	70	DKT3	10	Fourth diskette
31				71			
32				72			
33	DKT	10	Diskette	73	DKT1	10	Second diskette
34				74			
35				75			
36				76			
37				77	CPU	--	CPU and real-time clock

This page intentionally left blank.

APPENDIX B
OCTAL AND HEXADECIMAL CONVERSION

To convert a number from octal or hexadecimal to decimal, locate in each column of the appropriate table the decimal equivalent for the octal or hex digit in that position. Add the decimal equivalents to obtain the decimal number.

To convert a decimal number to octal or hexadecimal:

1. Locate the largest decimal value in the appropriate table that will fit into the decimal number to be converted;
2. Note its octal or hex equivalent and column position;
3. Find the decimal remainder.

Repeat the process on each remainder. When the remainder is 0, all digits will have been generated.

OCTAL CONVERSION TABLE

	8^5	8^4	8^3	8^2	8^1	8^0
0	0	0	0	0	0	0
1	32,768	4,096	512	64	8	1
2	65,536	8,192	1,024	128	16	2
3	98,304	12,228	1,536	192	24	3
4	131,072	16,384	2,048	256	32	4
5	163,840	20,480	2,560	320	40	5
6	196,608	24,576	3,072	384	48	6
7	229,376	28,672	3,584	448	56	7

HEXADECIMAL CONVERSION TABLE

	16^5	16^4	16^3	16^2	16^1	16^0
0	0	0	0	0	0	0
1	1,048,576	65,536	4,096	256	16	1
2	2,097,152	131,072	8,192	512	32	2
3	3,145,728	196,608	12,288	768	48	3
4	4,194,304	262,144	16,384	1,024	64	4
5	5,242,880	327,680	20,480	1,280	80	5
6	6,291,456	393,216	24,576	1,536	96	6
7	7,340,032	458,752	28,672	1,792	112	7
8	8,388,608	524,288	32,768	2,048	128	8
9	9,437,184	589,824	36,864	2,304	144	9
A	10,485,760	655,360	40,960	2,560	160	10
B	11,534,336	720,896	45,056	2,816	176	11
C	12,582,912	786,432	49,152	3,072	192	12
D	13,631,488	851,968	53,248	3,328	208	13
E	14,680,064	917,504	57,344	3,584	224	14
F	15,728,640	983,040	61,440	3,840	240	15

This page intentionally left blank.

APPENDIX C
ASCII CHARACTER CODES

DECIMAL	OCTAL (7-Bit)	HEX	ASCII SYMBOL	CONTROL FUNCTION	TO PRODUCE ON TTY Models 33 and 35			OCTAL 8-bit Code EVEN Parity
					CNTRL	SHIFT	Character	
0	000	00	NUL	Null	○	○	P	000
1	001	01	SOH	Start of Heading	○		A	201
2	002	02	STX	Start of Text	○		B	202
3	003	03	ETX	End of Text	○		C	003
4	004	04	EOT	End of Transmission	○		D	204
5	005	05	ENQ	Enquiry	○		E	005
6	006	06	ACK	Acknowledge	○		F	006
7	007	07	BEL	Bell	○		G	207
8	010	08	BS	Backspace	○		H	210
9	011	09	HT	Horizontal Tab	○		I	011
10	012	0A	NL	New Line			line feed	012
					○		J	012
					○		line feed	212[1]
11	013	0B	VT	Vertical Tab	○		K	213
12	014	0C	FF	Form Feed	○		L	014
13	015	0D	RT	Return			return	215
					○		M	215
					○		return	015[1]
14	016	0E	SO	Shift Out	○		N	216
15	017	0F	SI	Shift In	○		O	017
16	020	10	DLE	Data Link Escape	○		P	220
17	021	11	DC1	Device Control 1	○		Q	021
18	022	12	DC2	Device Control 2	○		R	022
19	023	13	DC3	Device Control 3	○		S	223
20	024	14	DC4	Device Control 4	○		T	024
21	025	15	NAK	Negative Acknowledge	○		U	225
22	026	16	SYN	Synchronous Idle	○		V	226
23	027	17	ETB	End Transmission Block	○		W	027
24	030	18	CAN	Cancel	○		X	030
25	031	19	EM	End of Medium	○		Y	231
26	032	1A	SUB	Substitute	○		Z	232
27	033	1B	ESC	Escape			esc	033
					○	○	K	033
28	034	1C	FS	File Separator	○	○	L	234
29	035	1D	GS	Group Separator	○	○	M	035
30	036	1E	RS	Record Separator	○	○	N	036
31	037	1F	US	Unit Separator	○	○	O	237
32	040	20	SP	Space			space	240

1. On even parity TTY's, these codes are odd parity.

APPENDIX C (Continued)

DECIMAL	OCTAL (7-Bit)	HEX	ASCII SYMBOL	TO PRODUCE on TTY Mod 33,35		OCTAL 8-Bit Code EVEN Parity	DECIMAL	OCTAL (7-Bit)	HEX	ASCII SYMBOL	TO PRODUCE on TTY Mod 33,35		OCTAL 8-Bit Code EVEN Parity
				SHIFT	Character						SHIFT	Character	
33	041	21	!	○	1	041	83	123	53	S		S	123
34	042	22	"	○	2	042	84	124	54	T		T	324
35	043	23	#	○	3	243	85	125	55	U		U	125
36	044	24	$	○	4	044	86	126	56	V		V	126
37	045	25	%	○	5	245	87	127	57	W		W	327
38	046	26	&	○	6	246	88	130	58	X		X	330
39	047	27	'	○	7	047	89	131	59	Y		Y	131
40	050	28	(	○	8	050	90	132	5A	Z		Z	132
41	051	29	)	○	9	251	91	133	5B	[	○	K	333
42	052	2A	*	○	:	252	92	134	5C	\	○	L	134
43	053	2B	+	○	;	053	93	135	5D	]	○	M	335
44	054	2C	,		,		94	136	5E	^	○	N	336
45	055	2D	-		-	055	95	137	5F	-	○	O	137
46	056	2E	.		.	056	96	140	60	`		`	140
47	057	2F	/		/	257	97	141	61	a			341
48	060	30	0		0	060	98	142	62	b			342
49	061	31	1		1	261	99	143	63	c			143
50	062	32	2		2	262	100	144	64	d			344
51	063	33	3		3	063	101	145	65	e			145
52	064	34	4		4	264	102	146	66	f			146
53	065	35	5		5	065	103	147	67	g			347
54	066	36	6		6	066	104	150	68	h			350
55	067	37	7		7	267	105	151	69	i			151
56	070	38	8		8	270	106	152	6A	j			152
57	071	39	9		9	071	107	153	6B	k			353
58	072	3A	:		:	072	108	154	6C	l			154
59	073	3B	;		;	273	109	155	6D	m			355
60	074	3C	<	○	,	074	110	156	6E	n			356
61	075	3D	=	○	-	275	111	157	6F	o			157
62	076	3E	>	○	.	276	112	160	70	p			360
63	077	3F	?	○	/	077	113	161	71	q			161
64	100	40	@	○	P	300	114	162	72	r			162
65	101	41	A		A	101	115	163	73	s			363
66	102	42	B		B	102	116	164	74	t			164
67	103	43	C		C	303	117	165	75	u			365
68	104	44	D		D	104	118	166	76	v			366
69	105	45	E		E	305	119	167	77	w			167
70	106	46	F		F	306	120	170	78	x			170
71	107	47	G		G	107	121	171	79	y			371
72	110	48	H		H	110	122	172	7A	z			372
73	111	49	I		I	311	123	173	7B	{			173
74	112	4A	J		J	312	124	174	7C	|			374
75	113	4B	K		K	113	125	175	7D	}			175
76	114	4C	L		L	314	126	176	7E	~			176
77	115	4D	M		M	115	127	177	7F	DEL		rubout	377
78	116	4E	N		N	116							
79	117	4F	O		O	317							
80	120	50	P		P	120							
81	121	51	Q		Q	321							
82	122	52	R		R	322							

APPENDIX D
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 two's complement notation. The high-order part of a negative number is therefore in one's complement form unless the low order part is null (at the right, only 0's are null regardless of sign). Hence, in processing double length numbers, two's complement operations are usually confined to the low order parts, whereas one's 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.

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

Note that the magnitude parts of the sequence of negative numbers from the most negative toward zero are the positive numbers from zero upward. 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    0,0
ADD    2,0
```

In two's 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 one's complement and a two's 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
```

This page intentionally left blank

APPENDIX E
INSTRUCTION USE EXAMPLES

On the following pages are examples of how the instruction set of the microNOVA computers may be used to perform some common functions.

- Clear an AC and the carry bit.

```
SUBO   AC,AC
```

- Clear and AC an preserve the carry bit.

```
SUBC   AC,AC
```

- Generate the indicated constants.

```
SUBZL    AC,AC     ;Generate +1
ADC      AC,AC     ;Generate -1
ADCZL    AC,AC     ;Generate -2
```

- Let ACN be any accumulator whose contents are zero; generate the indicated constants in ACN.

```
INCZL    ACN,ACN     ;Generate +2
INCOL    ACN,ACN     ;Generate +3
INCS     ACN,ACN     ;Generate +400 8
```

- Check if both bytes in an accumulator are equal.

```
MOVS     ACS,ACD
SUB      ACS,ACD,SZR
JMP      ---              ;Not equal
---      ---              ;Equal
```

- Check if two accumulators are both zero.

```
MOVS     ACS,ACS,SNR
SUB      ACS,ACD,SZR
JMP      ---              ;Not equal
---      ---              ;Equal
```

- Check an ASCII character to make sure it is a decimal digit. The character is in ACS and is not destroyed by the test. Accumulators ACx and ACy are destroyed.

```
LDA      ACx,C60          ;ASCII zero
LDA      ACy,C71          ;ASCII nine
ADCZ#    ACy,ACS,SNC      ;Skips if (ACS>9)
ADCZ#    ACS,ACx,SZC      ;Skips if (ACS≧0)
JMP      ---              ;Not digit
---      ---              ;Digit
C60:     60               ;ASCII 0
C71:     71               ;ASCII 9
```

- Test an accumulator for zero.

```
MOV      AC,AC,SZR
JMP      ---              ;Not zero
---      ---              ;Zero
```

- Test an accumulator for -1.

```
COM#     AC,AC,SNR
JMP      ---              ;Not -1
---      ---              ;-1
```

- Test an accumulator for 2 or greater.

```
MOVZR #    AC,AC,SNR
JMP        ---          ;Less than 2
---        ---          ;2 or greater
```

- Assume that it is known that AC contains 0, 1, 2, or 3; find out which value.

```
MOVZR #    AC,AC,SEZ
JMP        THREE        ;Was 3
MOV        AC,AC,SNR
JMP        ZERO         ;Was 0
MOVZR #    AC,AC,SZR
JMP        TWO          ;Was 2
---        ---          ;Was 1
```

- Perform the following unsigned integer comparisons.

```
SUB #      ACS,ACD,SZR    ;Skip if (ACS) = (ACD)
SUB #      ACS,ACD,SNR    ;Skip if (ACS)≠(ACD)
ADCZ #     ACS,ACD,SNC    ;Skip if (ACS) < (ACD)
SUBZ #     ACS,ACD,SNC    ;Skip if (ACS) ≤ (ACD)
SUBZ #     ACS,ACD,SZC    ;Skip if (ACS) > (ACD)
ADCZ #     ACS,ACD,SZC    ;Skip if (ACS) ≥ (ACD)
```

- Subtract 1 from an accumulator without using a constant from memory.

```
NEG        AC,AC
COM        AC,AC
```

- Multiply an AC by the indicated value.

```
MOV        ACx,ACx      ;Multiply by 1


MOVZL      ACx,ACx      ;Multiply by 2


MOVZL      ACx,ACy      ;Multiply by 3
ADD        ACy,ACx

ADDZL      ACx,ACx      ;Multiply by 4

MOV        ACx,ACy
ADDZL      ACx,ACx      ;Multiply by 5
ADD        ACy,ACx

MOVZL      ACx,ACy      ;Multiply by 6
ADDZL      ACy,ACx

MOVZL      ACx,ACy      ;Multiply by 7
ADDZL      ACy,ACy
SUB        ACx,ACy      ;in ACy

ADDZL      ACx,ACx      ;Multiply by 8
MOVZL      ACx,ACx

MOVZL      ACx,ACy      ;Multiply by 9
ADDZL      ACy,ACy
ADD        ACy,ACx

MOV        ACx,ACy      ;Multiply by 10
ADDZL      ACx,ACx
ADDZL      ACy,ACx

MOVZL      ACx,ACy      ;Multiply by 12
ADDZL      ACy,ACx
MOVZL      ACx,ACx

MOVZL      ACx,ACy      ;Multiply by 18
ADDZL      ACy,ACy
ADDZL      ACy,ACx
```

- Peform the inclusive OR of the operands in AC0 and AC1. The result is placed in AC1. The carry bit is unchanged.

```
COM        0,0
AND        0,1
ADC        0,1
```

- Perform the exclusive OR of the operands in AC0 and AC1. The result is placd in AC1. The contents of AC2 and the carry bit are destroyed.

```
MOV        1,2
ANDZL      0,2
ADD        0,1
SUB        2,1
```

- Assume that AC0 contains a signed, 16-bit, two's complement integer. The following three instructions will place an indicator of the sigh of the number in AC0. If the number is greater than 0, AC0 is set to +1. If the number is less than 0 AC0 is set to -1. If the number is equal to 0, AC0 remains 0. The previous contents of the carry bit are lost.

```
ADDO       AC0,AC0,SBN    ;Skip if GT 0
ADCC       AC0,AC0,SNC    ;AC0 gets -1
SUBCL      AC0,AC0        ;Copy carry into bit 15
```

- Move 30 words from locations 2000 - 2035_8 to locations 3000 - 3035_8. Two auto-increment locations are used to hold the source and destination addresses.

```
        LDA     0,ADDRS     ;Set up source address
        STA     0,20
        LDA     0,ADDRD     ;Set up destination address
        STA     0,21
LOOP:   LDA     0,@ 20      ;Increment source address and get word
        STA     0,@ 21      ;Increment destination address and store word
        DSZ                 ;Decrement count
        JMP     CNT         ;Go back for next word
        ---     LOOP        ;Skip here when count is zero
        ---
ADDRS:  1777                ;Source address minus one
ADDRD:  2777                ;Destination address minus one
CNT:    36                  Word count -- 36₈ equals 30₁₀
```

- Compare the signed two's complement integer contained in ACS to 0.

```
        MOV#     ACS,ACS,SZR   ;Skip if contents of ACS EQ 0
        MOV#     ACS,ACS,SNR   ;Skip if contents of ACS NE 0
        ADDO #   ACS,ACS,SBN   ;Skip if contents of ACS GT 0
        MOVL #   ACS,ACS,SZC   ;Skip if contents of ACS GE 0
        MOVL#    ACS,ACS,SNC   ;Skip if contents of ACS LT 0
        ADDO#    ACS,ACS,SEZ   ;Skip if contents of ACS LE 0
```

- Load a byte from memory. The routine is called via a JSR. The byte pointer for the requested byte is in AC2. The requested byte is returned in the right half of AC0. The left half of AC0 is set to 0. AC1, AC2, and the carry bit are unchanged. AC3 is destroyed.

```
LBYT:   STA     3,LRET      ;Save return address
        LDA     3,MASK
        MOVR    2,2,SNC     ;Turn byte pointer into word address and skip if
                            ; requested byte is right byte
        MOVS    3,3         ;Swap mask if requested byte is left byte
        LDA     0,0,2       ;Place word in AC0
        AND     1,0,SNC     ;Mask off unwanted byte and skip if swap is not
                            ; needed
        MOVS    0,0         ;Swap requested bte into right half of AC0
        MOVL    2,2         ;Restore byte pointer and carry
        JMP     @LRET       ;Return location
LRET:   0
MASK:   377
```

APPENDIX E (Continued)

- Store a byte in memory. The routine is called via a JSR. The byte to be stored is in the right half of AC0 with the left half of AC0 set to 0. The byte pointer is in AC2. The word written is returned in AC0. AC1, AC2, and the carry bit are unchanged. AC3 is destroyed.

```
SBYT:  STA    3,SRET   ;Save return
       STA    1,SAC1   ;Save AC1
       LDA    3,MASK
       MOVR   2,2,SNC  ;Convert byte pointer to word address
                       ; and skip if Byte is to be right half
       MOVS   0,0,SKP  ;Swap byte and leave mask alone
       MOVS   3,3      ;Swap mask
       LDA    1,0,2    ;Load word that is to receive byte
       AND    3,1      ;Mask off byte that is to receive new byte
       ADD    1,0      ;Add memory word on top of new byte
       STA    0,0,2    ;Store word with new byte
       MOVL   2,2      ;Restore byte pointer and carry
       LDA    1,SAC1   ;Restore AC1
       JMP    @SRET    ;Return
SRET:  0               ;Return location
SAC1:  0
MASK:  377
```

- The transfer of control between routines is made easier and more orderly by using the stack facility of the microNOVA computers.

  The basic method of transferring control to a subroutine is via a JUMP TO SUBROUTINE instruction. The subroutine executes a SAVE instruction at the subroutine entry point and returns control via the RETURN instruction.

```
          ;Calling program
CALL:     JSR              SUBR
          ---
          ---
          ---
SUBR:     SAV
          ---
          ---
          ---
RETRN:    RET
```

This method has the following characteristics:

1. AC3 of the calling program is destroyed by the JSR.

2. The call is only one word.

3. Upon return to the calling program, AC3 contains the calling program's frame pointer.

4. A SAVE instruction is required at each entry point.

5. Arguments are easily passed on the stack because SAVE sets up the frame pointer for the called routine and RETURN places the frame pointer of the calling routine in AC3.

APPENDIX F
INSTRUCTION EXECUTION TIMES

All times are in microseconds.

LDA, STA	2.9
ISZ, DSZ	3.8
JMP	2.9
JSR	3.4
ADD, SUB, NEG, INC	2.4
MOV, AND, COM, ADC	2.4
Each level of @, add:	1.0
Each autoindex, add:	2.4
Index register addr, add:	0.0
If skip occurs, add:	1.0
PSHA, POPA	3.4
SAV	7.7
RET	7.2
MUL	41.3
DIV	59.0
I/O input	7.2
I/O output	4.8
P.I. CYCLE	3.8
INTERRUPT LATENCY	N/A
DATA CHANNEL	
Input	6.7
Output	5.8
Latency	19.7

This page intentionally left blank.

CUT ALONG DOTTED LINE

READERS COMMENT FORM

DOCUMENT TITLE: ....

Your comments, accompanied by answers to the following questions, help us improve the quality and usefulness of our publications. If your answer to a question is "no" or requires qualification, please explain.

How did you use this publication?

( ) As an introduction to the subject.
( ) As an aid for advanced knowledge.
( ) For information about operating procedures.
( ) To instruct in a class.
( ) As a student in a class.
( ) As a reference manual.
( ) Other..

Did you find the material:

- Useful............................YES ( ) NO ( )
- Complete.......................YES ( ) NO ( )
- Accurate......................YES ( ) NO ( )
- Well organized...........YES ( ) NO ( )
- Well written.................YES ( ) NO ( )
- Well illustrated............YES ( ) NO ( )
- Well indexed...............YES ( ) NO ( )
- Easy to read...............YES ( ) NO ( )
- Easy to understand.....YES ( ) NO ( )

We would appreciate any other comments; please label each comment as an addition, deletion, change, or error and reference page numbers where applicable.

COMMENTS

PAGE	COL	PARA	LINE	FROM	TO

From:

NAME.. TITLE..................................
FIRM.. DIV. ....
ADDRESS..
CITY.. STATE.............. ZIP..........
TELEPHONE.............................. DATE..................................

Data General Corporation

ENGINEERING PUBLICATIONS
COMMENT FORM
DG-00935

FOLD DOWN FIRST FOLD DOWN

FIRST CLASS
PERMIT NO. 26
SOUTHBORO
MASS. 01772

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

Postage will be paid by:

DataGeneral
Southboro, Massachusetts 01772

ATTENTION: Engineering Publications

FOLD UP SECOND FOLD UP

STAPLE