NOVA® Line Computers

NOVA® 4
PROGRAMMER'S REFERENCE
MANUAL

NOTICE

CONTENTS

Chapter I
NOVA 4 SYSTEM

INTRODUCTION

The NOVA 4 is a general purpose, four-accumulator, stored-program computer, with a word length of 16 bits. The maximum amount of main memory is 64 Kbytes without a MAP and 256 Kbytes or 131,072 16-bit words with a MAP.

Memory can be addressed either directly or by using indirect addresses. A data channel is provided to enable rapid data transfer between main memory and peripheral devices.

The standard instruction set contains instructions that perform fixed point arithmetic and logical operations between accumulators, transfer of operands between accumulators and main memory, transfer of program control, and input/output (I/O) operations. Options are available that add instructions to this set. These additional instructions perform such operations as multiply/divide, floating point calculations, and memory allocation and protection.

Efficient Basic Instruction Set

The basic instruction set for the NOVA 4 contains instructions that perform fixed point arithmetic and logical operations between accumulators, transfer of operands between accumulators and main memory, 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.

Stack

A Last-In/First-Out (LIFO) or push-down stack is maintained by the NOVA 4 processor. This feature provides a convenient method for saving return information and passing arguments between subroutines. The stack also provides an expandable area for the temporary storage of variables and intermediate results.

Floating Point

The floating point feature allows the manipulation of both single precision (32 bits) and double precision (64 bits) floating point numbers. Single precision gives 6-7 significant decimal digits while double precision gives 15-17 significant decimal digits. The decimal range of a floating point number is approximately 5.4×10^{-79} to $7.2 \times 10^{+75}$ in either precision.

The floating point feature contains two 64-bit floating point accumulators. Floating point calculations can take place between these two accumulators or between one of the accumulators and operands in main memory.

Memory Allocation and Protection

The optional Memory Allocation and Protection unit (MAP) translates logical addresses within the user space to physical memory addresses. The MAP feature holds two user maps and two data channel maps at a time. Only one user map can be enabled at any one time, but both data channel maps are enabled at the same time.

In addition to translating addresses, the feature also performs various protection functions. A user is allowed to access only those blocks of memory allocated to him. This ensures that a user does not reach out of his own areas of memory for either instructions or data. Blocks of memory allocated to a

user may be write-protected so that the user may not modify them. This allows blocks of memory containing constants or non self-modifying procedures to be shared between users.

The MAP detects and inhibits indirection chains that go deeper than 16 levels. This protects the system from becoming disabled by an indirection loop. The MAP also provides I/O protection which allows all I/O devices to be declared accessible or inaccessible to a user.

Memory

Memory for the NOVA 4 is available in 32 Kbyte, 64 Kbyte, 128 Kbyte and 256 Kbyte modules. All memory is semiconductor.

Auto-Increment/Decrement

If the intermediate address of a short class instruction is in the range 20-27$_8$, and the indirect bit is 1, the contents of the addressed location are incremented by one. The incremented value is used to continue the addressing chain.

If the intermediate address of a short class instruction is in the range 30-37$_8$, and the indirect bit is 1, the contents of the addressed location are decremented by one. the decremented value is used to continue the addressing chain.

> NOTE: *The state of bit 0 before the increment or decrement determines whether the indirection chain is continued. For example: Assume an auto-increment location contains 177777$_8$ (all bits = 1 including bit 0), and the location is referenced as part of an indirection chain. After incrementing, the location contains all zeros. However, bit 0 was 1 before the increment, so 0 will be the next address in the chain rather than the effective address.*

Power Fail/Auto Restart

The power fail/auto restart feature of the NOVA 4 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.

When power is restored, the action taken by the auto-restart portion of the feature depends upon the position of the lock switch on the front panel. If the switch is not in the *lock* position, the processor remains stopped after power is restored. If the switch is in the *lock* position and battery backup is

operational, then after power is restored, the processor executes the instruction contained in the first location of main memory, restarting the interrupted system.

The battery backup option available with the NOVA 4 operates in conjunction with the power fail/auto restart feature to provide security for semiconductor memories 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 one-and-one-half hours (depending on the number of memory boards) so that they will not lose their data. If further security is desired, a larger external battery can be attached to ensure the integrity of the memories for extended periods of time.

Real-Time Clock

The real-time clock feature of the NOVA 4 generates a sequence of pulses that is independent of the timing of the processor. The clock will interrupt the system at one of four program-selectable frequencies. The frequencies are: ac line frequency, 10Hz, 100Hz, and 1000Hz.

Input/Output Bus

The input/output (I/O) bus is that portion of the computer that carries commands and data between the central processor and various peripheral devices connected to it. The bus is made up of a six-line device selection network, interrupt circuitry, command circuitry, and sixteen data lines.

Device Addressability

Each I/O device in a NOVA 4 is connected to the six-line device selection network in such a way that each device will only respond to commands that contain its own device code. The fact that the selection network is made up of six lines gives $2^6 = 64$ unique device codes. Ten of these codes are reserved for specific functions, but there are still 54 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 the routine can use a special instruction to identify the source of the interrupt.

The interrupt circuitry of the NOVA 4 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 greater transfer rates, the I/O bus contains circuitry for a data channel through which a device, at its own request, can gain direct access to main memory using a minimum of processor time. At the maximum transfer rate, the data channel effectively stops the processor, but at lower rates, processing continues while data is being transferred.

Ease of Interfacing

Due to the straightforward logic and general design of the NOVA 4 I/O bus, customer-provided or customer-designed I/O devices may be easily interfaced to a NOVA 4. Information on how to interface to the NOVA 4 may be found in "The Interface Designer's Reference Manual" (DGC 015-000031).

Input/Output Devices

A comprehensive array of I/O devices is available from Data General for the NOVA 4. This wide choice of devices, ranging from teletypewriters to line printers to video displays for man-machine interaction; and from paper tape to magnetic tape to fixed and moving-head discs for data storage allows a wide spectrum of possible configurations. Also available are various multiplexors and telecommunications adapters, including an IBM 360/370 interface.

Software

A wide variety of software support is available for the NOVA 4.

Operating systems include the Disc Operating System (DOS), the Real-Time Operating System (RTOS), and the Real-Time Disc Operating system (RDOS).

An assembler is available with all of these operating systems. In addition, many higher-level languages are available. These include Fortran IV and V, DG/L^{TM}, ALGOL, Extended BASIC, and Business BASIC. Note that not all languages are available in all operating systems.

I- 3

Chapter II
INTERNAL STRUCTURE

INTRODUCTION

The basic structure of a NOVA 4 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.

DG-05588
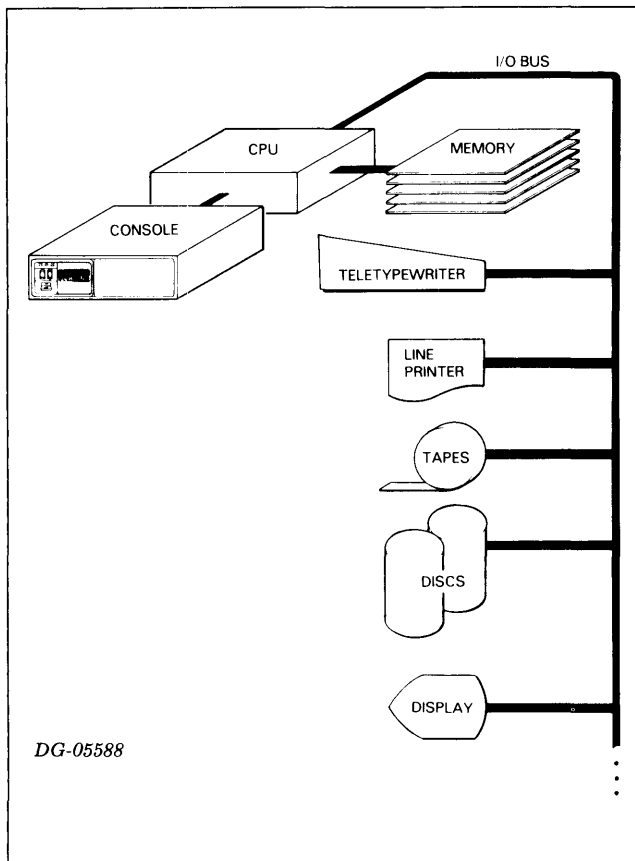
Due to the general-purpose design of the NOVA 4, the type, size, and number of memory modules and I/O devices have no effect upon the internal logical structure of the CPU. This chapter deals with the addressing of information and the logical representation of information within the CPU, and is unaffected 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 16 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 broken 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 (highorder) 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.

```
        WORD                     WORD
 ┌────────────────┐     ┌────────────────┐
    BYTE      BYTE          BYTE      BYTE
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```
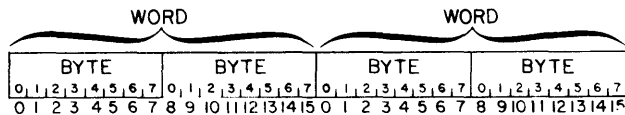
Octal Representation

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. Each encoded digit is called an octal digit. Because each group of bits can contain any one of 8 values, this representation is sometimes called *base 8* representation.

Another way to represent binary information is the hexadecimal or *hex* representaion. 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.

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. 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 the NOVA 4 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 the 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 three different, basic information formats. They are integers, floating point numbers, and logical quantities.

Integers

Integers can be represented as either signed or unsigned numbers and 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 43,294,967,295.

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 represents 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. 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.

II- 2

a true zero. Note that true zero is positive. It is not possible to obtain negative zero as the result of a calculation.

Logical Quanities

Logical operations in the NOVA 4 can be performed upon individual bits, bytes, or words. When using the logical operations, quantities operated on are treated as unstructured binary quantities. The number of bits, bytes, or words operated upon depends on the particular instruction.

INFORMATION ADDRESSING

The information formats described in the preceding section give a way of representing different types of data in main memory. 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.

DG 00538
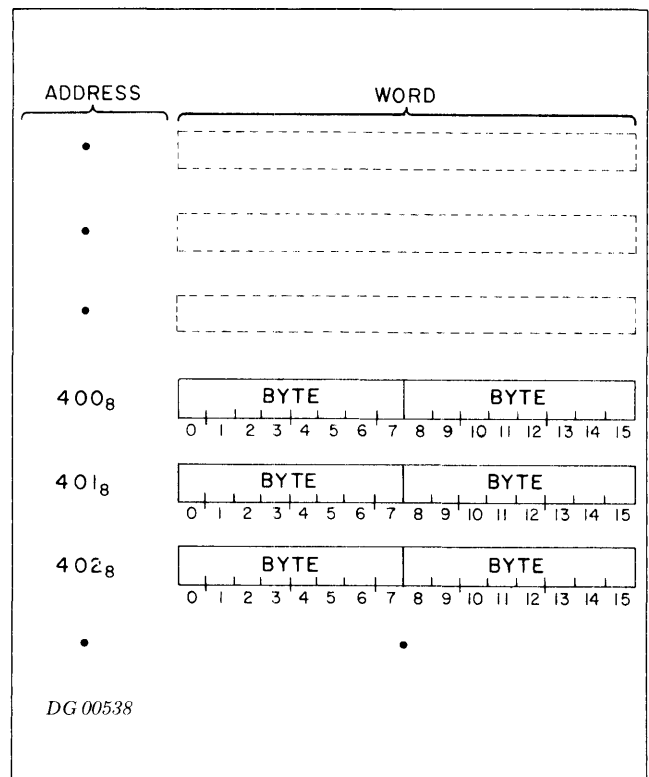
Effective Address Calculation

There are six instructions in the NOVA 4 instruction set that directly reference memory using word addressing. These instructions use eleven bits in the instruction to define 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 eleven bits in an instruction that are used in the effective address calculation, are bits 5-15. Their format is shown below.
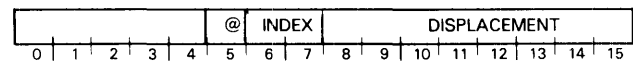
Bit 5 is called the *indirect bit*, bits 6 and 7 are called the *index bits*, and bits 8-15 are called the *displacement bits.*

If the index bits are 00, the displacement is used as an unsigned 8-bit number to address one of the first 256_{10} words in memory. This is called *page zero addressing* and this first block of 256 words is known as *page zero.*

If the index bits are 01, the displacement is treated as a signed, two's complement number, which is added to the address of the instruction to produce a memory address. This is called *relative addressing.* By relative addressing, any instruction which uses the effective address calculation can directly address any word in

II- 4

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 a 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

The floating point feature of the NOVA 4 allows operations 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 NOVA 4 floating point number. Since floating point numbers occupy either two words for single precision or four words for double precision, and the floating point feature is much faster than multiple precision integer software routines, 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

S	EXPONENT	MANTISSA

0 1   7 8   31

Double Precision

S	EXPONENT	MANTISSA

0 1   7 8   63
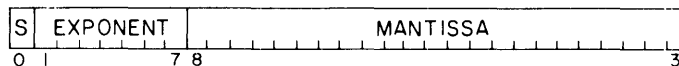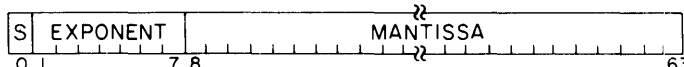
DG-05589

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*. When a calculation results in a zero mantissa, the floating point processor automatically converts the number to

II-3

storage whose address is in the range -128_{10} to $+127_{10}$ 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 index register does not change the value contained in the index register.

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 100006_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 zero, the address already computed is taken as the effective address. If the indirect bit is one, the word addressed by the result of the index and displacement bits is assumed to contain an address. 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 until a word is retrieved with bit 0 set to 0. Bits 1-15 of this word are taken to be the effective address.

Auto-Increment/Decrement

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 location), 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.

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.*

DG 00539
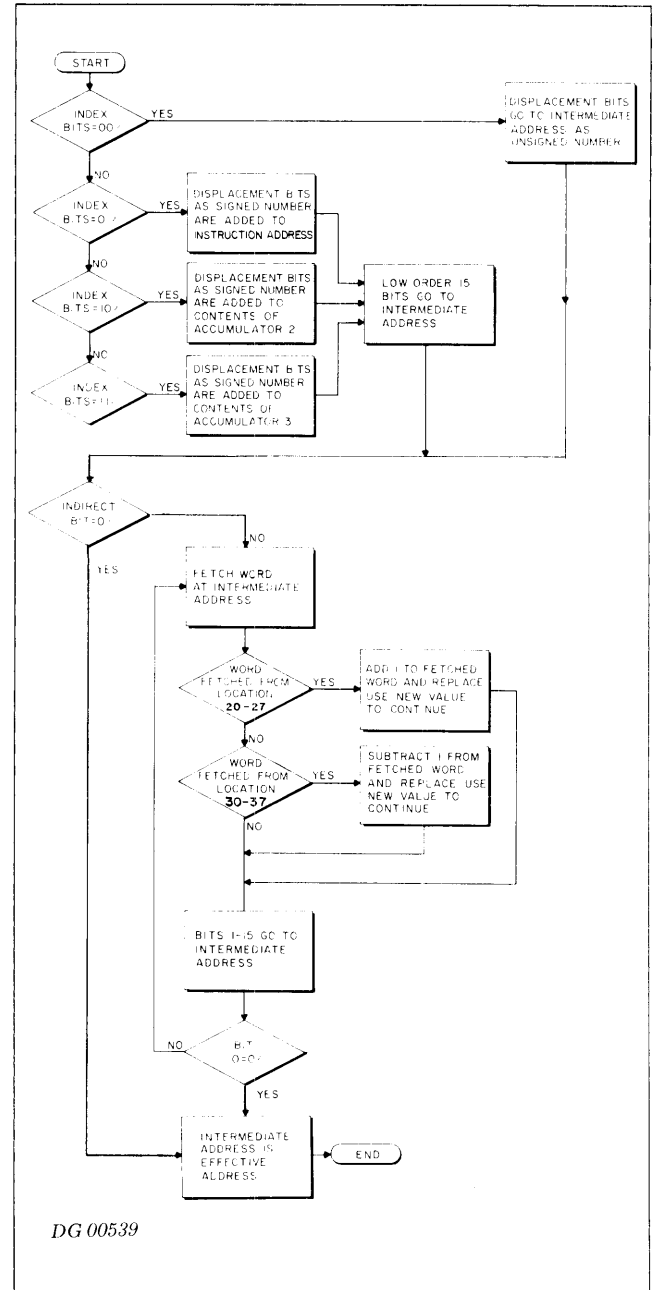
An effective address is always 15 bits in length. This means that an instruction which uses the effective address calculation can address any one of $32,768_{10}$ words. This gives rise to the concept of an *address space*, which, in the NOVA 4, contains 64K bytes or 32,768 2-byte words.

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

II-5

bits 0-14 are the address in memory of a 2-byte word. Bit 15 is the *byte indicator.* If the byte indicator is 0, the byte pointer references the high-order byte (bits 0-7) of the word in memory; if it is 1, the pointer references the low order byte (bits 8-15).

DG-00930
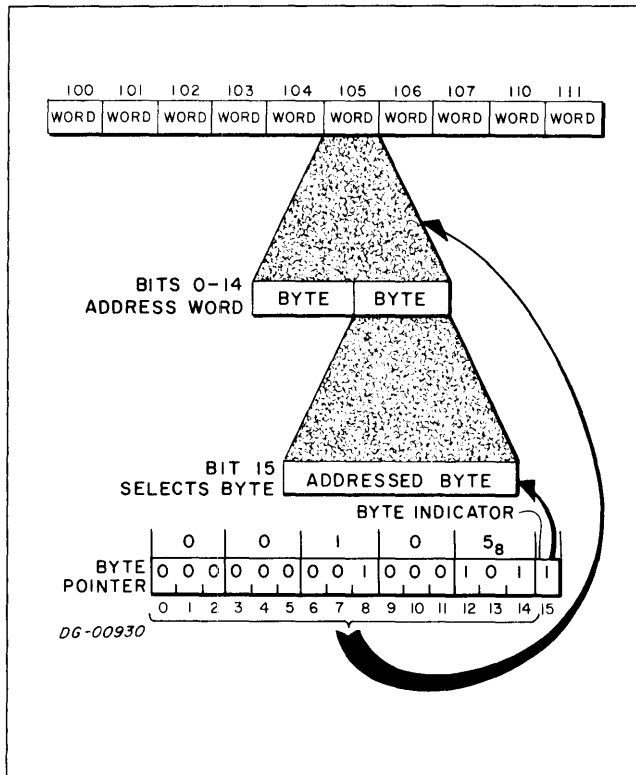
Addressing With Address Translation Hardware

The concept of an address space was introduced in the discussion of effective address calculation. The *program* or *logical* address space is that amount of memory that can be referenced by instructions in a program. The maximum logical address space available to a program running on a NOVA 4 is 64K bytes or 32K words.

The *physical* address space is that amount of physical memory that can be referenced by the CPU. If the MAP is not installed, the maximum physical address space available to the CPU is 64K bytes or 32K words, and the logical address space is equal to the physical space. For a NOVA 4 with the MAP installed, the maximum physical address space is 256K bytes and the logical address space is some subset of the physical space.

Installation of a MAP has no effect on logical addressing. Addressing calculations remain the same. The MAP translates the 15-bit address from the CPU into a 17-bit address and uses this new address to perform the memory reference.

PROGRAM EXECUTION

Programs for the NOVA 4 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 this address. This 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.

DG 00543
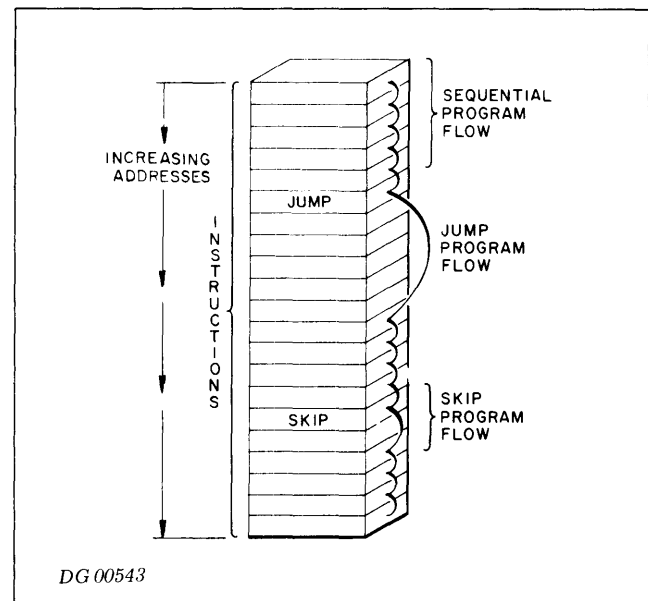
Because the program counter is 15 bits in length, it can address 32,768 separate memory locations. The next memory location after 77777_8 is location 0, and the location before 0 is location 77777_8. If the program counter rolls from 77777_8 to 0 in the course of sequential operation, no indication is 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 this case, 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 fault handler, sequential operation of the program resumes.

DG 00544

014-000617-01   PROGRAM EXECUTION
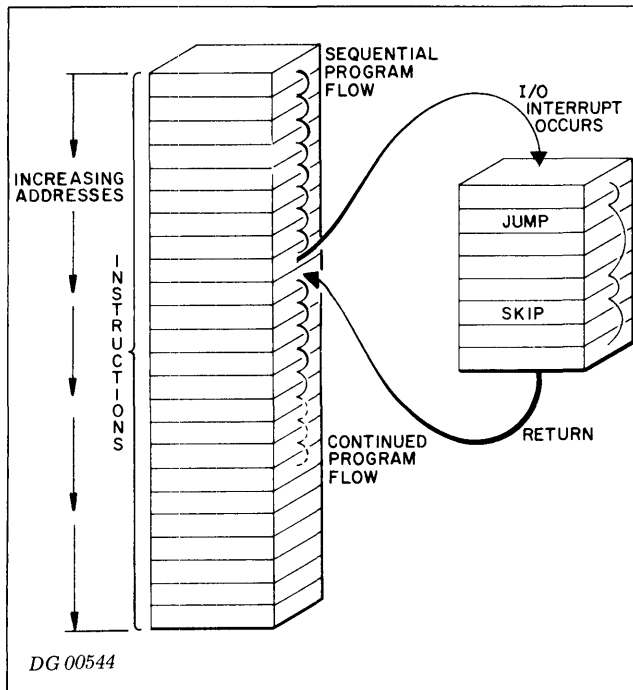
Chapter III

INSTRUCTIONS SETS

INTRODUCTION

The instruction set implemented on the NOVA 4 is divided into 5 sets. There are instruction sets available for fixed point arithmetic, logical operations, program flow alteration, floating point arithmetic, and I/O operations. In addition, instructions are available for programming the stack, MAP, the Real Time Clock, power fail/auto-restart, and certain CPU functions.

INSTRUCTION FORMATS

There are four different formats for instructions on the NOVA 4. 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.

0	0	0	OP'N	@	INDEX	DISPLACEMENT
0	1	2	3 4	5	6 7	8 9 10 11 12 13 14 15
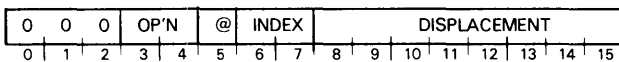
In the No Accumulator-Effective Address format instructions, bits 0-2 are 000, and bits 3-4 contain the operation code. The effective address is computed from bits 5-15 as described under *Effective Address Calculation.*

0	0	0	AC	@	INDEX	DISPLACEMENT
0	1	2	3 4	5	6 7	8 9 10 11 12 13 14 15
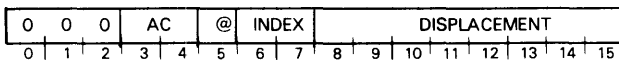
In the One Accumulator-Effective Address format instructions, bit 0 is 0, and bits 1-2 contain the operation code. Bits 3-4 specify the accumulator for the operation. The effective address is computed from bits 5-15 as described under *Effective Address Calculation.*

0	1	1	AC	OP CODE	F	DEVICE CODE
0	1	2	3 4	5 6 7	8 9	10 11 12 13 14 15
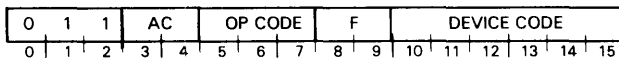
In the Input/Output format instructions, bits 0-2 are

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.

1	ACS	ACD	OP CODE	SH	C	#	SKIP
0	1 2	3 4	5 6 7	8 9	10 11	12	13 14 15
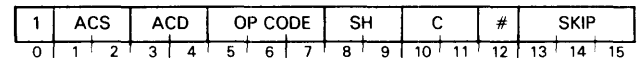
In the Arithmetic/Logical Class instructions, bit 0 is 1, bits 1 and 2 specify the source accumulator, bits 3 and 4 specify the destination accumulator, bits 5-7 contain the operation code, bits 8 and 9 specify the action of the shifter, bits 10 and 11 specify the value to which the carry bit will be initialized, bit 12 specifies whether or not the result will be loaded into the destination accumulator, and bits 13-15 specify the skip test.

ALC Instruction Execution

The ALC instructions use an Arithmetic Logic Unit (ALU) to process data. The logical organization of the ALU is illustrated below.

ORGANIZATION OF ARITHMETIC UNIT

DG-00927
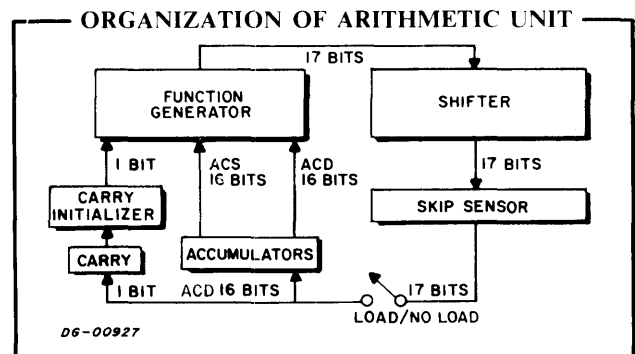
When an ALC instruction begins execution, it loads the contents of the carry bit and the contents of the accumulator(s) to be processed into the ALU. There are five distinct stages of ALU operation. We will discuss these stages separately.

Carry

The ALU begins its manipulation of the data by determining a new value for the carry bit. This new value is based upon three things: the old value of the carry, bits 10-11 of the ALC instruction, and the ALC instruction being executed. The ALU first determines the effect of the instruction bits 10-11 on the old value of the carry. The table below shows each of the mnemonics that can be appended to the instruction mnemonic, the value of bits 10-11 for each choice, and the action each one takes.

SYMBOL	VALUE	OPERATION
[c] omitted	00	Leave Carry bit unchanged
[c]=Z	01	Initialize Carry bit to 0
[c]=O	10	Initialize Carry bit to 1
[c]=C	11	Complement the Carry bit

Function

The ALU next evaluates the effect of the specific function (bits 5-7) upon the data. For the instructions *Move, AND,* and *Complement* the ALU performs the function on the data word(s) and saves the result. The value of the carry is as it was calculated above. For the instructions *Add, Add Complement, Subtract, Negate,* and *Increment* the result of the function's action upon the data word(s) may be larger than $2^{16} - 1$. A carry out results. In this situation, the ALU saves the low-order 16 bits of the function result, but it complements the value of the carry calculated above.

> NOTE: *At this stage of operation, the ALU does not load either the saved value of the function result into the destination accumulator, or the saved value of the carry into the carry bit.*

Shift Operations

Next the ALU performs any specified shift operation on the 17 bits output from the function generator (16 bits of data plus the calculated value of the carry bit). Depending on which shift operation is specified in the instruction, the function generator output can be rotated left or right one bit, or have its bytes swapped. The first table below shows the different shift operations that can be performed, the value of bits 8-9 for each choice, and the action each choice takes. The second table shows how each shift operation works.

SYMBOL	VALUE	OPERATION
[sh] omitted	00	Do not shift the result of the ALC operation
[sh]=L	01	Rotate left the 17-bit combination of Carry bit and ALC operation result
[sh]=R	10	Rotate right the 17-bit combination of Carry bit and ALC operation result
[sh]=S	11	Swap the two 8-bit halves of the ALC operation result without affecting Carry bit

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.
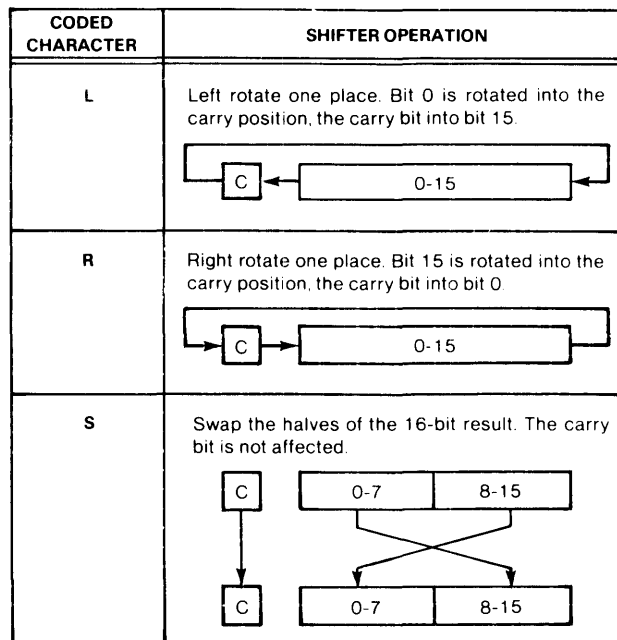
DG 04423

Skip Tests

The ALU can test the result of the shift operation for one of a variety of conditions, and skip or not skip the next instruction depending upon the result of the test. The table below shows the tests that can be performed, the value of bits 13-15 for each choice, and the action each choice takes.

SYMBOL	VALUE	OPERATION
[skip] omitted	000	No skip
[skip]=SKP	001	Skip unconditionally
[skip]=SZC	010	Skip if Carry bit is zero
[skip]=SNC	011	Skip if Carry bit is nonzero
[skip]=SZR	100	Skip if ALC result is zero
[skip]=SNR	101	Skip if ALC result is nonzero
[skip]=SEZ	110	Skip if either ALC result or Carry bit is zero
[skip]=SBN	111	Skip if both ALC result and Carry bit is nonzero

Load/No-Load

If the no-load bit (bit 12) is 0, the ALU loads the result of the shift operation into the destination accumulator, and loads the new value of the carry into the carry bit. If the no-load bit is 1, then the ALU does not load the result of the shift operation into the destination accumulator, and does not load the new value of the carry into the carry bit, but all other operations, such as skip tests, take place. This no-load option is particularly convenient to use when you want to test for some condition without destroying the contents of the destination accumulator. The table below shows how to code the load/no-load operation.

SYMBOL	VALUE	OPERATION
# omitted	0	Load the result of the shift operation into ACD
#	1	Do not load the ALC operation result into ACD; restore Carry bit to value it had before shifting

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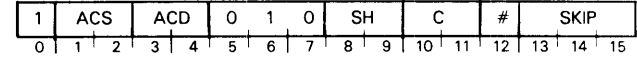
To shift the number in AC3 one bit left without destroying the number, and skip the next sequential instruction if the bit shifted into the carry bit is zero, the following instruction could be coded:

MOVL# 3,3,SZC

This instruction assembles 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

III- 3
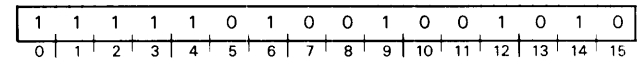
CODING AIDS

We use certain conventions and abbreviations throughout this chapter to help you properly code each instruction for Data General's assembler. Briefly, they are these:

[] *[]* Square brackets indicate that the enclosed symbol (e.g., *[,skip])* is an optional operand or mnemonic. Code it only if you want to specify the option.

BOLD Code operands or mnemonics printed in boldface exactly as shown. For example, code the mnemonic for the *Move* instruction: **MOV**.

italic For each operand or mnemonic in italics, replace the item with a number or symbol that provides the assembler value you need for that item (e.g., the proper accumulator number, an address, etc.).

We use the following abbreviations throughout this chapter:

ABBR	MEANING
AC	Accumulator
ACS	Source accumulator
ACD	Destination accumulator
FPAC	Floating point accumulator

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

The indirect bit (bit 5) is set to 1 by coding the symbol @ anywhere in the effective address 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. If no index is coded, the bits are set to 00. The character *period* (.) can be used to set the index bits to 01. *Period* can be read to mean *address of the current instructions.* When the period is used, it is followed by either a plus or a 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 programmer supplies numbers to the assembler. The default radix is Base 8 or octal. The assembler radix can be changed by using the RADIX statement.

The assembler available with the NOVA 4 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.

FIXED POINT ARITHMETIC

The fixed point instruction set performs binary arithmetic on operands in accumulators. The operands are 16 bits in length and can be either signed or unsigned. The instruction set provides for loading, storing, adding, and subtracting.

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
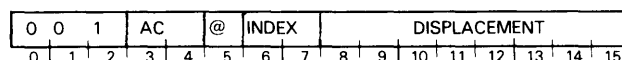
Copies a word from memory to an accumulator.

Places the word addressed by the effective address, *E*, 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
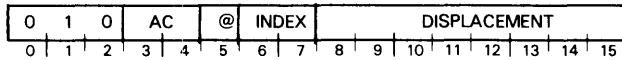
Stores the contents of an accumulator into a memory location.

Places the contents of the specified accumulator 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.

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
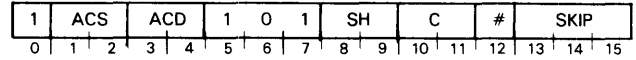
Performs unsigned integer subtraction and complements the carry bit if appropriate.

Initializes the carry bit to its specified value. The instruction subtracts the unsigned, 16-bit number in ACS 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 instruction places the result of the addition in the shifter. If the operation produces a carry of 1 out of the high-order bit, the instruction complements the carry bit. The instruction performs the specified shift operation and places the result of the shift in ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

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

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

Performs unsigned integer addition and complements the carry bit if appropriate.

Initializes the carry bit to the specified value, adds the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The instruction then performs the specified shift operation and places the result of the shift 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, the instruction complements the Carry bit.*

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
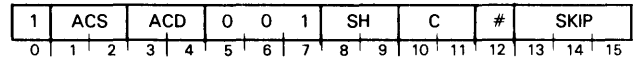
Forms the two's complement of the contents of an accumulator.

Initializes the carry bit to the specified value. Places the two's complement of the unsigned, 16-bit number in ACS in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the instruction complements the carry bit. Performs the specified shift operation and places the result in ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

NOTE: *If ACS contains 0, the instruction complements the carry bit.*

Add Complement

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

1	ACS		ACD		1	0	0	SH		C		#	SKIP		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
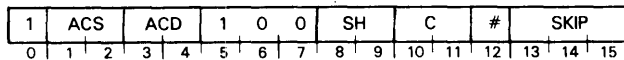
Adds an unsigned integer to the logical complement of another unsigned integer.

Initializes the carry bit to the specified value, adds the logical complement of the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The instruction then performs the specified shift operation, and loads the result of the shift 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 instruction complements the Carry bit.*

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
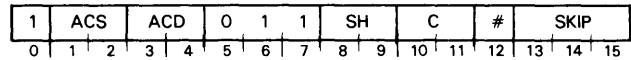
Increments the contents of an accumulator.

Initializes the carry bit to the specified value. Increments the unsigned, 16-bit number in ACS by one and places the result in the shifter. If the incrementation produces a carry of 1 out of the high order bit, the instruction complements the carry bit. Performs the specified shift operation, and loads the result of the shift 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 instruction complements the carry bit.*

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
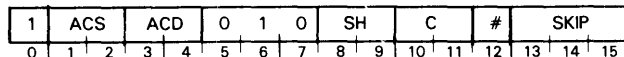
Moves the contents of an accumulator through the Arithmetic Logic Unit (ALU).

Initializes the carry bit to the specified value. Places the contents of ACS in the shifter. Performs the specified shift operation and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

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: *And*, and *Complement.*

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

Forms the logical AND of the contents of two accumulators.

Initializes the carry bit to the specified value and places the logical AND of ACS and ACD 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 resulting bit is 0. The instruction then performs the specified shift operation and places the result in ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

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

Forms the logical complement of the contents of an accumulator.

Initializes the carry bit to the specified value, forms the logical complement of the number in ACS, and performs the specified shift operation. The instruction then places the result 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 NOVA 4 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 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.

DG 00561
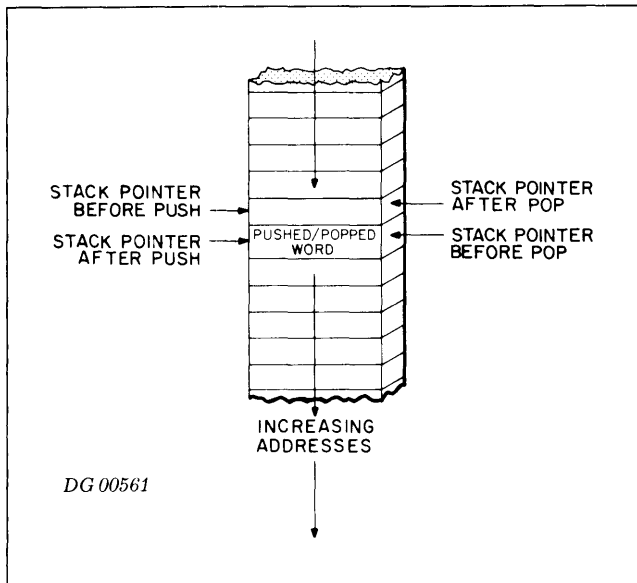
Frame Pointer

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 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 the return block is as follows:

WORD 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:

DG 00566
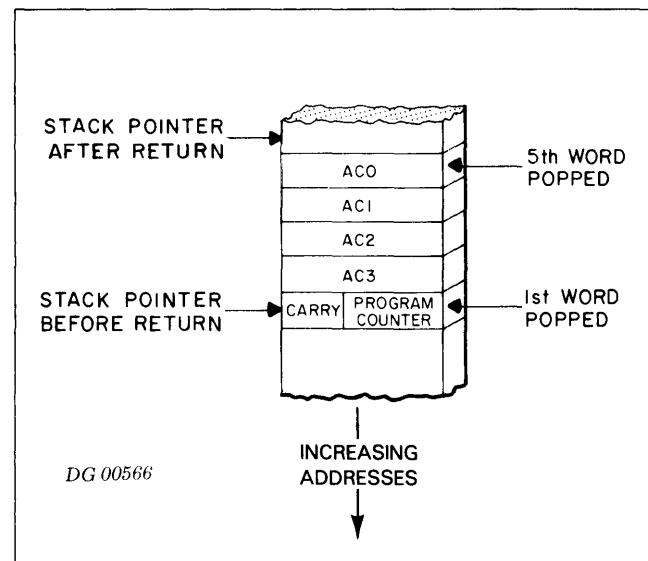
Stack Frames

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 calls. Variables and arguments can be transmitted from the calling routine to the called routine by placing them in prearranged

positions in the stack frame of the calling routine. 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 stack frome of the calling routine 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: if a program map is enabled, it is inhibited; the Interrupt On flag is set to 0; the stack overflow flag is set to 0; the updated program counter is stored in physical location 0; and the processor executes a *jump indirect* to physical location 3.

Initialization of the Stack Control Registers

Before the first operation on the stack can be performed, the stack control registers 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.

STACK MANIPULATION INSTRUCTIONS

The stack feature of the NOVA 4 computer 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	1	AC	0	1	1	0	0	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
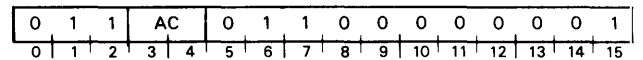
Pushes the contents of the specified accumulator onto the stack, and increments the stack pointer by one.

III- 9

Pop Accumulator

POPA   *ac*

0	1	1	AC	0	1	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
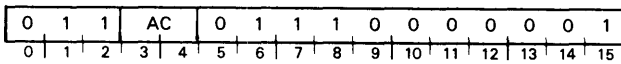
Pops 1 word off the stack, places it in the indicated accumulator, and decrements the stack pointer by one.

Move To Stack Pointer

MTSP   *ac*

0	1	1	AC	0	1	0	0	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
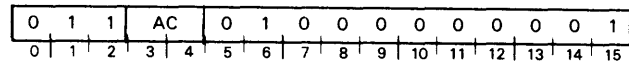
Places bits 1-15 of the specified accumulator in the stack pointer. The contents of the specified accumulator remain unchanged.

Save

SAV

0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
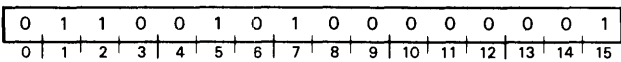
Pushes a return block onto the stack. After the fifth word of the return block is pushed, the value of the stack pointer is placed in the frame pointer and in AC3. The format of the five words pushed is as follows:

WORD 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 Frame Pointer

MTFP   *ac*

0	1	1	AC	0	0	0	0	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
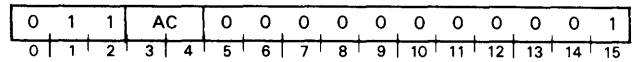
Places bits 1-15 of the specified accumulator in the frame pointer. The contents of the specified accumulator remain unchanged.

Move From Stack Pointer

MFSP *ac*

0	1	1	AC		0	1	0	1	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
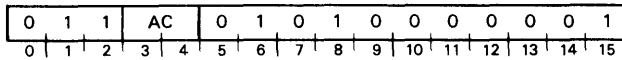
Places the contents of the stack pointer in bits 1-15 of the specified accumulator. Sets bit 0 of the accumulator to 0. The contents of the stack pointer remain unchanged.

Move From Frame Pointer

MFFP *ac*

0	1	1	AC		0	0	0	1	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
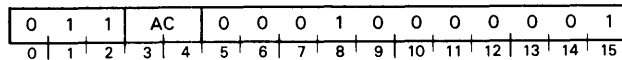
Places the contents of the frame pointer in bits 1-15 of the specified accumulator. Sets bit 0 of the accumulator to 0. The contents of the specified accumulator remain unchanged.

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 operations 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 instruction if the result is zero.

Jump

JMP

0	0	0	0	0	@	INDEX		DISPLACEMENT							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
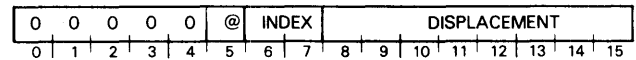
Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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
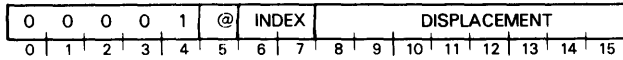
Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

Computes the effective address, *E*; then places the address of the next sequential instruction in AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

NOTE: *The instruction computes* E *before it places the incremented program counter in AC3.*

Increment And Skip If Zero

ISZ   *[@]displacement[,index]*

0	0	0	1	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 9 10 11 12 13 14 15
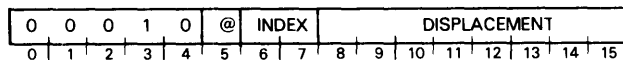
Increments the addressed word, then skips if the incremented value is zero.

Increments the word addressed by *E* and writes the result back into memory at that location. If the updated value of the location is zero, the instruction skips the next sequential word.

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
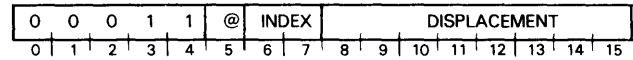
Decrements the addressed word, then skips if the decremented value is zero.

Decrements by one the word addressed by *E* and writes the result back into that location. If the updated value of the location is zero, the instruction skips the next sequential word.

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
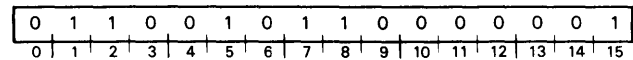
Places the contents of the frame pointer in the stack pointer and pops five words off the stack, placing them in the following locations:

WORD # POPPED	DESTINATION
1	Bit 0 placed in the carry bit. Bits 1-15 placed in the program counter.
2	Bits 1-15 placed in the frame pointer. Bits 0-15 placed in AC3
3	AC2
4	AC1
5	AC0

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

III-12

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
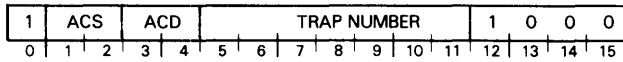
Disables the user map if enabled. Then places the logical address of this instruction in bits 1-15 of physical location 46_8, sets bit 0 of this location to 0, and jumps indirect via location 47_8. The state of the Interrupt On flag is unaltered.

BYTE INSTRUCTIONS

The following instructions move bytes to or from memory locations. Note that when an instruction moves a byte from memory to an accumulator, it also clears the high-order half of the destination accumulator. When an instruction moves a byte from an accumulator to memory, it leaves unchanged the other byte contained in that word of memory.

Load Byte

LDB   *acs,acd*

0	1	1	ACD		0	0	1	ACS		0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
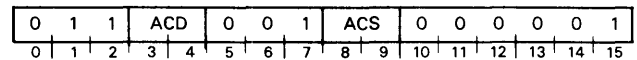
Moves a byte from memory (as addressed by a byte pointer in one accumulator) to the second accumulator.

Places the 8-bit byte addressed by the byte pointer contained in ACS in bits 8-15 of ACD. Sets bits 0-7 of ACD to 0. The contents of ACS remain unchanged unless ACS and ACD are the same accumulator.

Store Byte

STB *acs,acd*

0	1	1	ACD	1	0	0	ACS	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
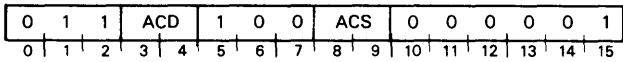
Moves the right byte of one accumulator to a byte in memory. The second accumulator contains the byte pointer.

Places bits 8-15 of ACD in the byte addressed by the byte pointer contained in ACS. The contents of ACS and ACD remain unchanged.

STB

Chapter 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 certain processor options, and perform certain processor functions.

The NOVA 4 has 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 each 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 Busy and Done are both 0, the device is idle and cannot perform any operations. To start a device, the program must set Busy to 1 and set Done to 0. When a device has finished its operation, it sets Busy to 0 and Done to 1. The case of Busy and Done both set to 1 is a meaningless situation and will produce unpredictable results.

The format for the I/O instructions is illustrated below.

0	1	1	AC	OP'N		ſ	DEVICE CODE
0	1	2	3 4	5 6 7	8 9	10 11 12 13 14 15	
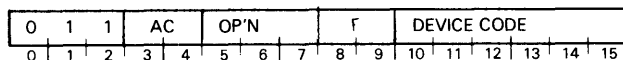
Bits 0-2 are 011, bits 3-4 specify the AC, bits 5-7 contain the operation code, bits 8-9 control the Busy and Done flags in the device, and bits 10-15 specify the code of the device. The six bits provided for the device code in the I/O format mean that 64 unique device codes are available for use. Some 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 available in Appendix A.

OPERATION OF I/O DEVICES

In general, the operation of all I/O devices is done by manipulation of 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 Busy to 1 and Done to 0. Once a device has completed its operation, and set Busy to 0 and Done 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 use the interrupt system that is standard on the NOVA 4. 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. An interrupt is initiated by an I/O device when it completes its operation.

Upon completing the operation, the device sets Busy to 0 and Done 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 Busy to 0 and Done 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 request 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 serivce routine. If no program map is enabled, the CPU places the updated program counter in physical memory location 0 and executes a *jump indirect* to physical memory location 1. It is assumed that location 1 contains the address, either direct or indirect, of the interrupt service routine. If the optional MAP is installed, it is inhibited; the updated program counter is placed in physical memory location 0 and the CPU executes a *jump indirect* to physical memory location 1.

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, 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.

The *Interrupt acknowledge* instruction returns the 6-bit device code of the device requesting the interrupt. If more than one device is requesting service, the code returned 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 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 to execute an *Interrupt enable* instruction, a JMP 0 instruction, and control will be returned to the interrupted program.

PRIORITY INTERRUPTS

If the Interrupt On flag remains 0 through 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 *Interrupt Acknowledge* 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 fixed head disc, the programmer may wish to set up a multiple level interrupt scheme. Hardware and instructions are available that allow the implementation of 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 the state of the machine upon receiving control. The state of the machine 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. The interrupt routine must determine which device requires service and jump 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 state of the machine, 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 an interrupt plus the execution of several instructions for each word transferred. To allow greater transfer rates, the NOVA 4 contains a data channel through which a device, at its own request, can gain direct access to memory, using a minimum of processor time.

When a device is ready to send or receive data, it requests access to memory 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, so 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. The data channel is fully described in the *Programmer's Reference Manual for Peripherals*, DGC number 015-000021.

CODING AIDS

We use certain conventions throughout this chapter to help you properly code each instruction for Data General's assembler. Briefly, they are:

[] *[]* Square brackets indicate that the enclosed symbol (e.g., *[,skip]* is an optional operand or mnemonic. Code it only if you want to specify the option.

BOLD Code operands or mnemonics printed in boldface exactly as shown. For example, code the mnemonic for the *Move* instruction: **MOV**.

italic For each operand or mnemonic in italics, replace the item with a number or symbol that provides the assembler value you need for that item (e.g., the proper accumulator number, an address, etc.).

We use the following abbreviations throughout this chapter:

f or **F** Device Flag Command

AC or AC Accumulator

The I/O instructions have optional mnemonics that can be appended to the standard mnemonic. These optional mnemonics control the Busy and Done flags of the I/O device addressed by the instruction. They are described in the following table.

SYMBOL	VALUE	OPERATION
[f] omitted	00	Does not alter the Busy and Done flags
[f]=**S**	01	Starts the device; Sets Busy flag to 1 Sets Done flag to 0
[f]=**C**	10	Idles the device; Sets Busy flag to 0 Sets Done flag to 0
[f]=**P**	11	I/O pulse; effect, if any depends on the device

The *I/O Skip* instruction allows you to test the state of the Busy and Done flags. You can perform any one of four tests by appending an optional mnemonic to the **SKP** mnenomic. The optional mnemonics are shown in the following table.

SYMBOL	VALUE	OPERATION
[t]=**BN**	00	Tests Busy flag for nonzero
[t]=**BZ**	01	Tests Busy flag for zero
[t]=**DN**	10	Tests Done flag for nonzero
[t]=**DZ**	11	Tests Done flag for zero

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
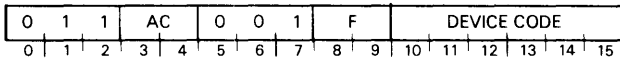
Transfers data from the A buffer of an I/O device to an accumulator.

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 0.

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
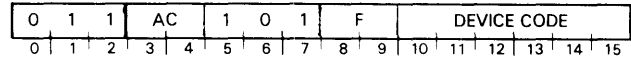
Transfers data from the C buffer of an I/O device to an accumulator.

Places the contents of the C input buffer in the specified device in the specified AC. After the data transfer, sets the Busy and Done flags according to the specified 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 0.

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
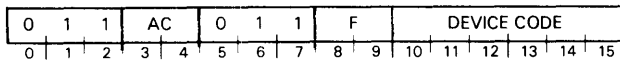
Transfers data from the B buffer of an I/O device to an accumulator.

Places the contents of the B input buffer in the specified device in the specified AC. After the data transfer, sets the Busy and Done flags 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 0.

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
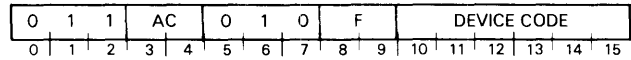
Transfers data from an accumulator to the A buffer of an I/O device.

Places the contents of the specified AC in the A output buffer of the specified device. After the data transfer, sets the Busy and Done flags 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	0	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 11 12 13 14 15
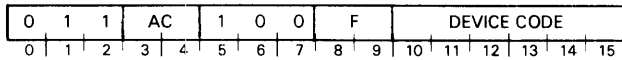
Transfers data from an accumulator to the B buffer of an I/O device.

Places the contents of the specified AC in the B output buffer of the specified device. After the data transfer, sets the Busy and Done flags 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
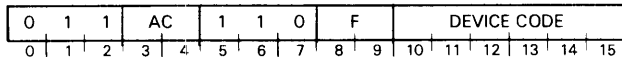
Transfers data from an accumulator to the C buffer of an I/O device.

Places the contents of the specified AC in the C output buffer of the specified device. After the data transfer, sets the Busy and Done flags 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
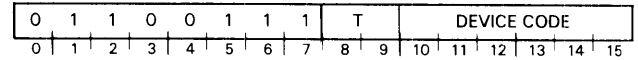
If the test condition specified by T is true, the instruction skips the next sequential word.

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
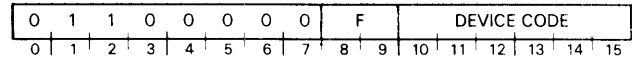
Used when a Busy or Done flag must be changed with no other operation taking place.

Sets the Busy and Done flags in the specified device according to the function specified by F.

CENTRAL PROCESSOR FUNCTIONS

I/O instructions with a device code of 77 perform a number of special functions rather than controlling a specific device. Device code 77 has been given the mnemonic CPU. In all but the *I/O Skip* instruction, I/O instructions with a device code of 77 use bits 8-9 to control the condition of the Interrupt On flag. An *I/O Skip* instruction with a device code of 77 uses bits 8-9 to either test the state of the Interrupt On flag or to test the state of the Power Fail flag. The mnemonics are the same as for the normal I/O instructions. The table below gives the result of these bits for instructions with a device code of 77.

IV-5

SYMBOL	VALUE	OPERATION
[f] omitted	00	Does not alter the Interrupt On flag
[f]=S	01	Sets Interrupt On flag to 1
[f]=C	10	Clears Interrupt On flag to 0
[f]=P	11	Leaves Interrupt On flag unchanged
[t]=BN	00	Tests Interrupt On flag for nonzero
[t]=BZ	01	Tests Interrupt On flag for zero
[t]=DN	10	Tests Power Fail flag for nonzero
[t]=DZ	11	Tests Power Fail flag for zero

Special Mnemonics

Some of the NOVA 4 I/O instructions which use device code 77 have special mnemonics which can be used in place of the standard mnemonics. Note that the mnemonics for controlling the state of flags cannot be appended to these special instruction mnemonics.

Thus, if you want to alter the state of the Interrupt On flag while performing a *Mask Out* instruction, you must use the full mnemonic:

   DOBf ac,CPU

instead of the special mnemonic:

   MSKO ac

The special mnemonic sets bits 8 and 9 to 00.

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
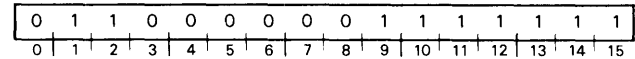
Sets Interrupt On flag to 1.

If the instruction changes the state of the Interrupt On flag, the CPU allows one more instruction to execute before the first I/O interrupt can occur. However, if the instruction is interruptible, then interrupts can occur as soon as the instruction begins to execute.

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
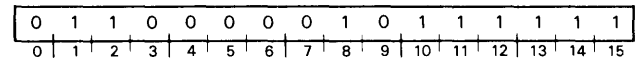
Sets Interrupt On flag to 0.

Read Switches

READS *ac*
DIA *[f]*   *ac*,CPU

0	1	1	AC	0	0	1	F	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
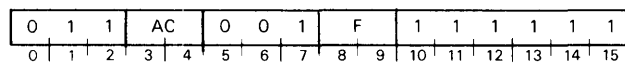
Places the contents of the virtual console switch register into an accumulator.

Places the contents of the virtual console switch register in the specified accumulator. After the transfer, sets the Interrupt On flag according to the function specified by F.

NOTE: *See Chapter VI for more information about the virtual console.*

Mask Out

MSKO
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

Sets the priority mask.

Places the contents of the specified accumulator in the priority mask. After the transfer, sets the Interrupt On flag according to the function specified by F. The contents of the specified AC remain unchanged.

NOTE: *A 1 in any bit disables interrupt requests at devices which use that bit as a mask.*

NOTE: *Do not use this instruction when interrupts are enabled.*

Interrupt Acknowledge

INTA
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

Returns device code of an interrupting device.

Places the six-bit device code of that device requesting an interrupt which is physically closest to the CPU on the I/O bus in bits 10-15 of the specified accumulator; sets bits 0-9 to 0. After the transfer, sets the Interrupt On flag according to the function specified by F.

Reset

IORST
DIC *[f]*   *ac*,CPU

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

Sets all Busy and Done flags and the priority mask to 0.

Sets the Busy and Done flags in all I/O devices to 0. Sets the 16-bit priority mask to 0. Sets the Interrupt On flag according to the function specified by F.

NOTE: *The assembler recognizes the mnemonic* IORST *as equivalent to the instruction* DICC 0,CPU.

If the mnemonic DIC *is used to perform this function, you must code an accumulator to avoid assembly errors. During execution, the accumulator field is ignored and the contents of the accumulator remain unchanged.*

Halt

HALT
DOC *[f]* *ac*,**CPU**

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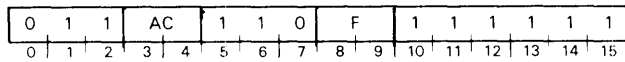
Stops the processor.

Sets the Interrupt On flag according to the function specified by F, then stops the processor and transfers control to the virtual console.

(to enable return to the interrupted program), put a *Jump* to the desired restart location in location 0, and then execute a *Halt*. One to two milliseconds is more than enough time to perform the power fail routine.

When power is restored, the action taken by the automatic restart portion of the power fail facility depends upon the position of the lock switch on the front panel. If the switch is not in the *lock* position, the CPU remains stopped after power is restored. If the switch is in the *lock* position and battery backup is operational, than after power is restored, the CPU executes a **JMP 0** instruction, restarting the interrupted program. If the switch is in the *lock* position and battery backup is *not* operational, then after power is restored the CPU transfers control to the virtual console.

The power fail facility has no device code and no interrupt disable bit in the priority mask. It does not respond to the *Interrupt Acknowledge* instruction. The Power Fail flag can be tested by the *CPU Skip* instruction.

CPU Skip

SKP *[t]* **CPU**

0	1	1	0	0	1	1	1	T		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
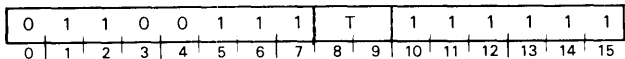
If the test condition specified by T is true, the next sequential word is skipped.

See *Programmer's Reference-Peripherals (DGC No. 015-000021)* for a complete set of examples on using the interrupt system.

POWER FAIL

In the NOVA 4, when power is turned off and on again, the state of the accumulators, the program counter, and the various flags in the CPU is indeterminate. The power fail facility, along with the optional battery-backup facility, 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 facility senses the imminent loss of power, sets the Power Fail flag, and requests an interrupt. The interrupt service routine can then use this delay to store the contents of the accumulators, the carry bit, and the current priority mask. The interrupt service routine should also save location 0

CPU Skip If Power Fail Flag Is One

SKPDN CPU

0	1	1	0	0	1	1	1	1	0	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
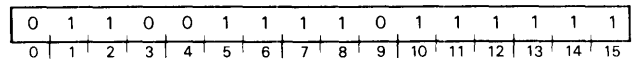
If the Power Fail flag is 1 (i.e., power is failing), the instruction skips the next sequential word.

IV- 8

CPU Skip If Power Fail Flag Is Zero

SKPDZ CPU

0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
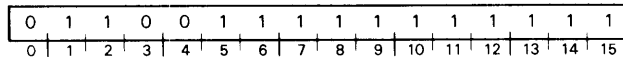
If the Power Fail flag is 0 (i.e., power is not failing), the instruction skips the next sequential word.

REAL-TIME CLOCK

The Real-Time Clock (RTC) facility on the NOVA 4 generates a sequence of pulses that is independent of the CPU timing. It will generate I/O interrupts at any one of four program selectable frequencies. The Busy and Done flags of the RTC are controlled by bits 8-9 of the I/O instruction. The RTC is device code 14_8 and has the mnemonic RTC. The interrupt disable bit is priority mask bit 13.

Setting Busy allows the next pulse from the clock to set Done, and the RTC requests an I/O interrupt if its interrupt disable bit is 0. A *Data out A* instruction to select the clock frequency only has to be given once. After each interrupt, an **NIOS** instruction will set up the clock for the next interrupt.

When Busy is first set, the first interrupt can come at any time up to the clock period. After the first interrupt has occurred, succeeding interrupts come at the clock frequency, provided that the program always sets Busy before the clock period expires. After power up or I/O reset, the clock is set to the line frequency.

The RTC frequency is selected by the following instruction:

Select RTC Frequency

DOA*[f]*   ac,RTC

0	1	1	AC	0	1	0	F	0	0	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
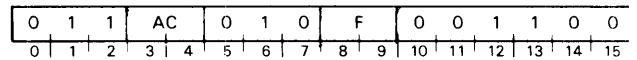
The clock frequency is set according to bits 14-15 of the specified AC. The contents of the specified AC remain unchanged. Bits 0-13 of the specified AC are ignored. The format of the specified AC is as follows:

														RTC
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
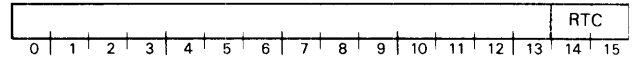
BITS	NAME	CONTENTS or FUNCTION
0-13	---	Reserved for future use. (Set to 0)
14-15	RTC	Selects the clock frequency as follows: 00 ac line frequency 01 10Hz 10 100Hz 11 1000Hz

IV-9

Chapter V
PROCESSOR OPTIONS

INTRODUCTION

In this chapter, we discuss the optional equipment for the NOVA 4. This includes the Multiply/Divide option, the Floating Point option, and the Memory Allocation and Protection unit (MAP).

MULTIPLY/DIVIDE

Multiplication can be performed on the NOVA line by software routines that utilize the standard instruction set, but if many of these operations are required, a loss of efficiency can result. The multiply/divide option provides the capability of performing these operations in firmware and hardware, with a corresponding increase in speed.

The multiply/divide option is part of the CPU. For compatibility, the instructions for the option are I/O instructions that reference device code 1. The assembler recognizes the mnemonics MUL and DIV for unsigned multiply and divide, and MULS and DIVS for signed multiply and divide.

Unsigned Multiply

MUL
DOCP   2,MDV

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
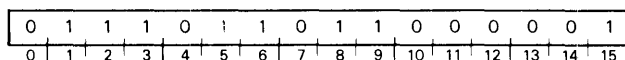
Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

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.

Signed Multiply

MULS
DOCC   3,MDV

0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
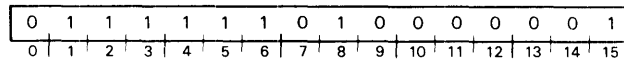
Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

The signed, 16-bit two's complement number in AC1 is multiplied by the signed, 16-bit two's complement number in AC2 to yield a signed, 32-bit two's complement intermediate result. The signed, 16-bit two's complement number in AC0 is added to the intermediate result to produce the final result. The final result is a signed, 32-bit two's complement number which occupies AC0 and AC1. Bit 0 of AC0 is the sign 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.

Unsigned Divide

DIV
DOCS   2,MDV

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
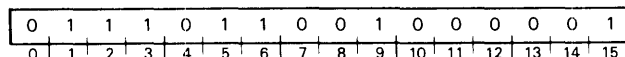
Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

Divides the unsigned 32-bit number contained in AC0 and AC1 by the unsigned, 16-bit number in AC2. 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.*

Signed Divide

DIVS
DOC   3,MDV

0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
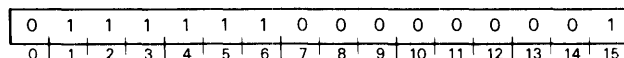
Divides the signed 32-bit integer in two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

The signed, 32-bit two's complement number contained in AC0 and AC1 is divided by the signed, 16-bit two's complement number in AC2. The quotient and remainder are signed, 16-bit numbers and occupy AC1 and AC0, respectively. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is always the same as the sign of the dividend, except that a zero quotient or a zero remainder is always positive. The carry bit is set to 0. The contents of AC2 remain unchanged.

> NOTE: *If the magnitude of the quotient is such that it will not fit into AC1, an overflow condition is indicated. The carry bit is set to 1, and the operation is terminated. The contents of AC0 and AC1 are unpredictable.*

MEMORY MANAGEMENT

Background to Address Translation

The concept behind the various memory management feature available with the NOVA 4 is that of *Logical-to-Physical Address Translation.* The amount of memory required by a user's program is defined to be his *logical address space.* This space may be as large as 32 2-Kbyte pages. The areas of physical storage assigned to the user are defined to be his *physical address space.* The address translation function that converts addresses in the logical space to addresses in the physical space is called the *address map* for that user. Each user has his own, unique logical-to-physical address map. In addition, there is a map for the data channel which can be, but does not have to be equal to the user map. The multiprogramming operating system determines what these maps are to be, and then transmits this information to the address translation hardware. The following illustration shows a possible two-user configuration.

This figure shows a 128Kword physical address space and its utilization by a two-user multiprogramming system. The supervisor resides in pages 0-7 of physical space. The first 16 pages of user #1 reside in pages 40-55 of physical space. User #2 also has his 32K of logical space split into two areas. Pages 0-15 of user #2 are in pages 24-39 of physical space and pages 16-31 of user #2 are in pages 56-71 of physical space. The data channel is capable of servicing both users. Any data channel reference to pages 0-15 of logical space will be mapped to pages 0-15 of the logical space of user #1. Any data channel reference to logical pages 16-31 will be mapped to pages 0-15 of the logical space of user #2.

DG 00233
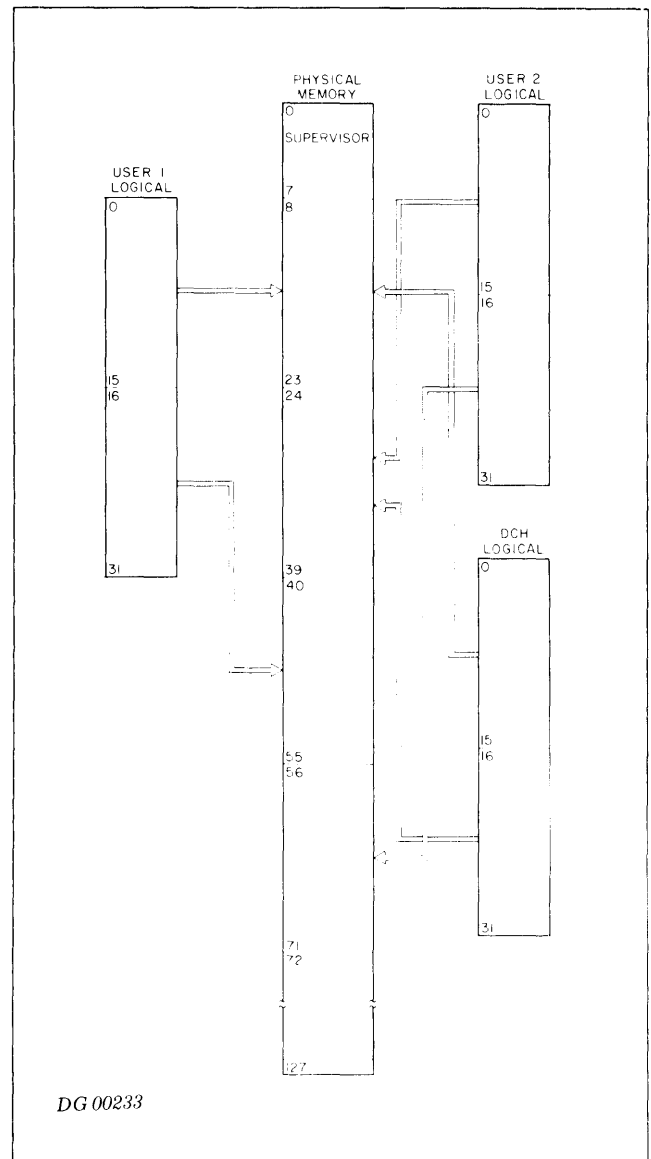
In order to manage memory efficiently, the operating system makes use of the validity and write protect features of the address translation hardware, if possible. The next figure shows a two-user configuration where these features are used.

In the second figure, a "W" in a page means the page is write-protected. By convention, mapping a logical page to physical page 127 and write protecting it makes that page validity protected. Both users have declared that page 1 of their logical space is to be write-protected.

Physical page 8 is the logical page 1 for user #1 and physical page 10 is the logical page 1 for user #2. User #1 is only using 13 pages of his 32 page logical address space, so logical pages 13-31 have been declared invalid for him. Any reference by user #1 to logical pages 13-31 will cause a validity error. User #2 is only using 21 pages of his logical space, so logical pages

V- 3

21-31 of his logical space have been declared invalid. Any reference by user #2 to logical pages 21-31 will result in a validity error.

The address translation hardware resides between the memory and the CPU, and the memory and the data channel, and is transparent to all of them. When either the CPU or the data channel requests a memory operation, the address translation hardware intercepts and services the request. The address translation hardware translates the 15 bit logical address coming from the CPU or the data channel into a 17 bit physical address. The memory operation is then performed using this 17 bit address. The memory access cycle time is unchanged.

The mapping information needed to service a CPU or data channel request is given to the address translation hardware by the operating system through I/O instructions that reference the address translation hardware. This information is transmitted before the supervisor enables either the user map or the data channel map.

DG 00232
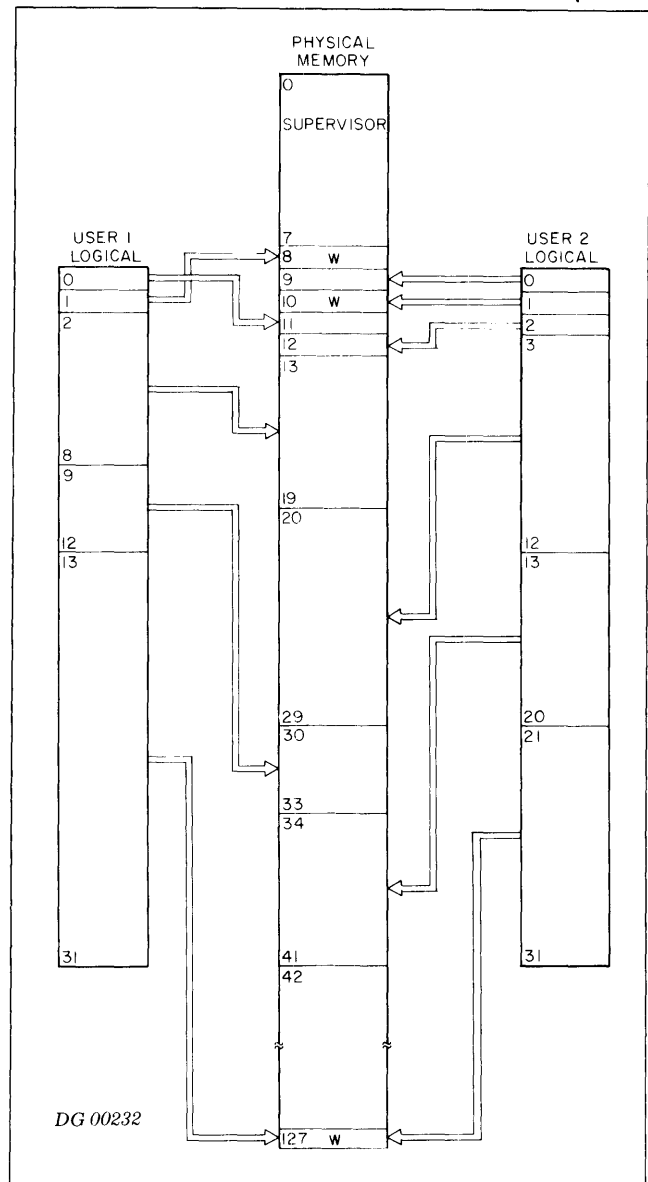
MEMORY ALLOCATION AND PROTECTION

The NOVA 4 is available with a Memory Allocation and Protection unit (MAP) which provides memory mapping and protection features.

The MAP expands the physical address space of a **NOVA 4 to 512Kbytes** by performing logical-to-physical address translation. The maximum logical address space is 64Kbytes. The MAP allows four maps (two user maps and two data channel maps) to be defined at any one time. These maps are called user map *A*, user map *B*, DCH map *A*, and DCH map *B*. Each map consists of 32 2-Kbyte pages. The selection of which user map is to be used to map logical addresses coming from the CPU is under program control. The selection of which data channel map is to be used is under control of the peripheral controllers. Those peripheral controllers not equipped to make this distinction will use data channel map *A* by default.

The two user maps and the two data channel maps are completely independent. Only one user map may be enabled at a time, but both data channel maps are enabled at the same time. The mapping of program addresses and mapping of data channel addresses may or may not be enabled at the same time depending upon the supervisor program. If either user mapping or data channel mapping is disabled then, for that function, the physical address space is equal to the logical address space and only the lowest 64 Kbytes of memory are accessible.

The instructions for the MAP are in the standard I/O format. The MAP takes two device codes: 2 and 3. The mnemonic for device code 2 is **MAP**. The mnemonic for device code 3 is **MAP1**.

Device code 2 has a Done flag which is set to 1 by the MAP any time address translation is enabled and not inhibited. Device code 2 also has a Busy flag which is set when a Data Channel error occurs. Device code 3 does not have a Busy or a Done flag.

The flag control commands for device code 2 are as follows:

f=S   Reserved for future use.

f=C   Clear violation status word.

f=P   The second non-data channel memory address after the issuance of this command is mapped using the map indicated by the Single Cycle Select bit in the MAP status word.

IORST   No effect.

The flag control commands for device code 3 are as follows:

f=S   Reserved for future use.

f=C   Disables the user map and data channel map portions of the MAP. Initializes all internal MAP logic. For diagnostic use only.

f=P   Reserved for future use.

IORST   No effect.

See the table under I/O Coding Aids for bit patterns of the flag control commands.

Load MAP

DOB*[f]*   ac,MAP

0	1	1	AC	1	0	0	F	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
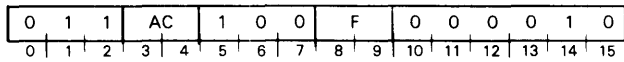
Transfers the contents of the specified AC to the MAP. Leaves the contents of the specified AC unchanged. The format of the AC is as follows:

SEL	LOGICAL	A/B	WP	VP	PHYSICAL
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
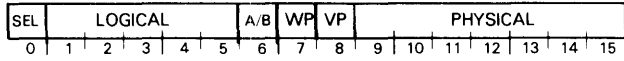
BITS	NAME	CONTENTS or FUNCTION
0	SEL	Specifies which type of map will be loaded by this instruction, as follows: 0 User map 1 Data channel map
1-5	Logical	Logical page number. This is a number in the range $0\text{-}37_8$.
6	A/B	Specifies map A or map B of the type specified by bit 0 as follows: 0 Map A 1 Map B
7	WP	If 1, write protection is enabled for this page.
8	VP	If 1, validity protection is enabled for this page (if bits 7 and 9-15 = 1).
9-15	Physical	Physical page number. This is a number in the range $0\text{-}177_8$.

Initiate page check

DOA*[f]*   ac,MAP1

0	1	1	AC	0	1	0	F	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
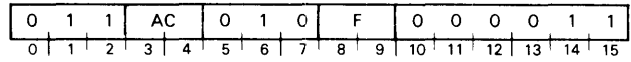
Transfers the contents of the specified AC to the MAP for later use by the *Page check* instruction. Leaves the contents of the specified AC unchanged. The format of the AC is as follows:

SEL	LOGICAL	A/B
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
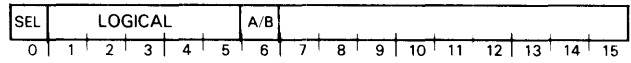
BITS	NAME	CONTENTS or FUNCTION
0	SEL	Selects type of map as follows: 0 User map 1 Data channel map
1-5	Logical	Logical page number. This is a number in the range $0\text{-}37_8$ and is the number of the logical page being checked.
6	A/B	Selects map A or Map B of the type selected by bit 0, as follows: 0 Map A 1 Map B
7-15	---	Reserved for future use. Should be 0.

Page check

DIA *[f]*   *ac,*MAP1

0	1	1	AC	0	0	1	F	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
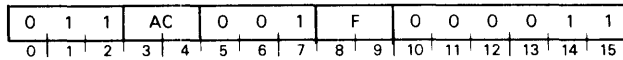
Places the physical page number corresponding to the logical page number specified in the last *Initiate page check* instruction into bits 9-15 of the specified AC. The format of the AC is as follows:

SEL	LOGICAL	A/B	WP	VP	PHYSICAL
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
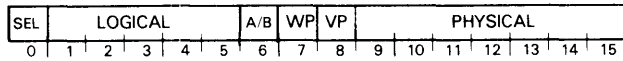
BITS	NAME	CONTENTS or FUNCTION
0	SEL	Map select bit from last *Initiate page check* instruction.
1-5	Logical	Logical page number from last *Initiate page check* instruction.
6	A/B	A/B select bit from last *Initiate page check* instruction.
7	WP	If 1, write protection is enabled for this page.
8	VP	If 1, validity protection is enabled for this page (if bits 7 and 9-15 = 1).
9-15	Physical	Physical page number corresponding to logical page number in bits 0-6.

Read MAP status

DIA *[f]*   *ac,*MAP

0	1	1	AC	0	0	1	F	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
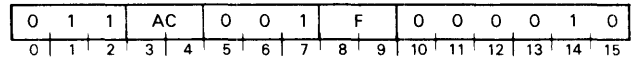
Places the 16-bit MAP status word in the specified AC. The format of the AC is as follows:

U/M	DCH	PMI							SWP	SAB	AIP	IND	I/O	WP	A/B
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
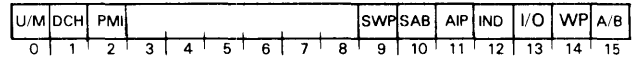
BITS	NAME	CONTENTS or FUNCTION
0	UM	If 1, user mapping is enabled.
1	DCM	If 1, data channel mapping is enabled.
2	PMI	If 1, user mapping is inhibited. Takes precedence over bit 0.
3-8	---	Reserved for future use. Set to 0.
9	SC/WP	If 1, single cycle write protection is enabled.
10	SC/AB	Single cycle map. 0 — Map A; 1 — Map B
11	AIP	If 1, auto increment/decrement protection is enabled.
12	IND	If 1, defer protection is enabled.
13	I/O	If 1, I/O protection is enabled.
14	WP	If 1, write protection is enabled.
15	A/B	User map select bit: 0 — Map A; 1 — Map B

Write MAP Status

DOA*[f]* *ac*,ac,MAP

0	1	1	AC	0	1	0	F	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
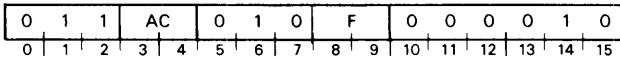
Places the contents of the specified accumulator in the MAP status word and sets the User Map Inhibit bit in the MAP status word to 0.

The instruction changes some of the bits in the MAP status word immediately, and changes others at the next indirect memory reference. The table below shows these differences.

The format of the specified AC is as follows:

U/M	DCH								SWP	SAB	AIP	IND	I/O	WP	A/B
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
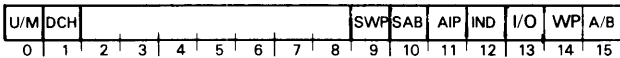
BITS	NAME	CONTENTS or FUNCTION
0	UM*	If 1, user mapping will be enabled.
1	DCH	If 1, data channel mapping will be enabled.
2-8	---	Reserved for future use. Should be set to 0.
9	SC/WP	If 1, single cycle write protection will be enabled.
10	SC/AB	Selects the map for single cyle operations as follows: 0 Map A 1 Map B
11	AIP*	If 1, auto increment/decrement protection will be enabled.
12	IND*	If 1, indirect protection will be enabled.
13	I/O*	If 1, I/O protection will be enabled.
14	WP*	If 1, write protection will be enabled.
15	A/B	Selects user map as follows: 0 Map A 1 Map B

*These bits are set at the next indirect memory reference.

Read Violation Data

DIB*[f]* *ac*,MAP

0	1	1	AC	0	1	1	F	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
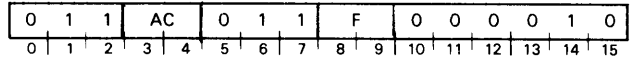
Places the MAP violation status bits in the specified AC, along with the logical page in which the violation occurred. The format of the AC is as follows:

ANY	LOGICAL								SC	VV	AIV	IND	I/O	WP	A/B
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
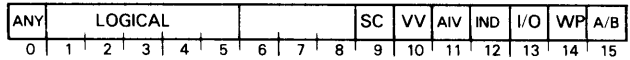
BITS	NAME	CONTENTS or FUNCTION
0	ANY	Violation flag. Set to 1 if any of bits 10-14 = 1.
1-5	Logical	Number of logical page in which violation occurred.
6-8	---	Reserved for future use. Set to 0.
9	SC	If 1, single cycle was enabled during violation.
10	VV	If 1, validity violation.
11	AIV	If 1, auto increment/decrement violation.
12	IND	If 1, defer violation.
13	I/O	If 1, I/O violation.
14	WP	If 1, write violation.
15	A/B	Map specifier bit: 0 Map A 1 Map B

MAP

Read Violation Address

DIB[f]   ac,MAP1

0	1	1	AC	0	1	1	F	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
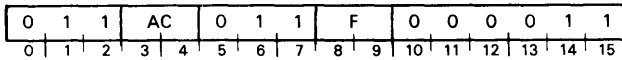
Places the logical address of the instruction that caused the violation in bits 1-15 of the specified AC. Sets bit 0 to zero.

Clear Violation

NIOC   MAP

0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
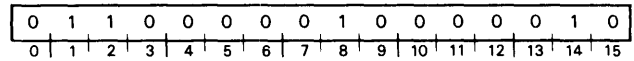
Clears the MAP violation status word and the Busy flag for device code 2, which indicates data channel errors.

Map Single Cycle

NIOP   MAP

0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
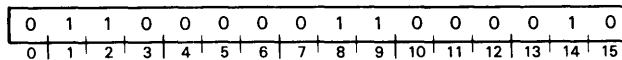
Enables the user map for one memory reference and maps the first memory reference of the next LDA or STA instruction. After the memory cycle is mapped, the instruction returns the map to the state it was in when the *Map Single Cycle* instruction was executed.

NOTE: *The* LDA *or* STA *instruction cannot use an indirect memory reference, and must* immediately *follow the* Map Single Cycle *instruction.*

Validity errors do trap; reads return ∅, writes are ignored.

Clear MAP

NIOC   MAP1

0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
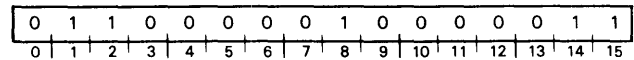
Disabled the user and data channel maps and initializes all internal MAP logic.

NOTE: *For diagnostic use only.*

SUPERVISOR PROGRAMMING FOR THE NOVA 4

Setting Up for Translation

The information that allows the MAP to translate addresses comes from the multiprogramming supervisor. The instructions are *Load MAP* and *Write MAP status* .

By using the *Load MAP* instruction, the supervisor gives the MAP a physical address for the beginning of a page of logical address space. Thirty-two *Load MAP* instructions are required to completely define the map for one logical space.

Although the floating point processor available with the NOVA 4 is an I/O device and operates through the data channel, all floating point operations are processed using the currently enabled user map.

After defining the maps that will be used, the supervisor gives the MAP information regarding how and when the maps are to be enabled via the *Write MAP status* instruction. This instruction also defines which protect features are to be enabled. Each protect feature described in the *Write MAP status* instruction can be enabled separately and independently of the others.

If a *Write MAP status* instruction is issued with bit 0 of the specified accumulator set to 1, then address translation will begin with the memory reference after the next defer cycle. This provides a convenient method for the supervisor to transfer control to the user program after the maps have been defined. One way of transferring this control is as follows:

```
        ...          ;ENOUGH LOAD MAP
        ...          ; INSTRUCTIONS TO
        ...          ; DEFINE ALL THE
        ...          ; MAPS THAT WILL
        ...          ; BE USED.
    LDA  0,STAT      ;
    DOA  0, MAP      ;WRITE MMPU STATUS
        ...          ;RESTORE USER'S
        ...          ; ACCUMULATORS.
        ...          ;--USE NO
        ...          ; INDIRECTION.
    INTEN            ;ENABLE INTERRUPTS
    JMP  @USERPC     ;ADDRESS IN
                     ; USERPC WILL BE
                     ; MAPPED.
STAT: 140000           ;STATUS WORD:
                     ; ENABLE USER
                     ; MAPPING,
                     ; ENABLE DCH
                     ; MAPPING.
                     ; SINGLE CYCLE MAP
                     ; FOR USER A
                     ; MAP ADDRESSES
                     ; FOR USER A.
USERPC:                ;STARTING ADDRESS.
```

Note that a defer instruction must appear after the *Write MAP status* instruction and before the next *Write MAP status* instruction for the second instruction to take effect.

MAP Protection Processing

When a map violation is detected, interrupts are inhibited, and address translation is disabled. The contents of physical location 46_8 are lost and the supervisor directs the CPU to *jump indirect* to location 47_8. The supervisor can then determine the type of violation using the *Read violation data* instruction.

The *Read violation address* instruction can be used to find the instruction that caused the problem.

> NOTE: *Location 46_8 is normally where the return address is found after a* Trap *instruction has been executed. If the trap is caused by a MAP violation, however, location 46_8 should be ignored and the* Read violation *instruction used instead.*

The MAP performs checking only for those protection features that are enabled. The five types of protection and how they are handled by the MAP are discussed below.

I/O Protection

If I/O protection is enabled in the NOVA 4 MAP, it protects all I/O devices except those using device codes 1, 74, 75, and 76. Device code 1 is generally assigned to the NOVA 4 multiply/divide option, and device codes 74-76 are generally assigned to the optional Floating Point Unit. The I/O devices using these device codes are not protected by I/O protection under any circumstances.

When I/O protection is enabled, the MAP decodes all I/O instructions to see if the referenced device is user protected. If it is, the MAP does not allow the execution of the instruction. Instead, it stores the logical address of the instruction in the *Violation address* register, disables I/O interrupt requests, enters the supervisor mode, and directs the CPU to *jump indirect* to location 47_8.

Validity Protection

By convention, validity protection cannot be disabled. A logical page is validity protected by mapping the page to physical page $255_{10}(377_8)$, and setting the validity protect and write protect bits to 1.

> NOTE: *It is not necessary for physical page 255_{10} to exist. Validity protection is indicated by setting the physical page bits to 377_8, and setting the write protect bit to 1. Since validity protection prevents the writing of the page, the existence of the physical page is not required.*

The MAP checks all CPU requests for invalid addresses. If the address is found to be valid, the MAP proceeds with the required translation. If the address is invalid, the MAP stores the logical address of the instruction in the *Violation address* register. The MAP then disables I/O interrupt requests, enters the supervisor mode, and directs the CPU to *jump indirect* to location 47_8.

Runaway Defer Protection

If runaway defer protection is enabled, the MAP checks memory references to see if they are part of a defer cycle. If the MAP detects 15 consecutive defer cycle memory requests, it traps.

Upon receiving the 15 requests, the MAP stores the address of the instruction that started the defer loop in the *Violation address* register. The MAP then disables the I/O interrupt requests, enters the supervisor mode, and directs the CPU to *jump indirect* to location 47_8.

Write Protection

If write protection is enabled, the MAP monitors all modify memory requests and determines whether or not that logical page is write-protected. If the page is not write-protected, the MAP allows the operation to proceed. If the page is write-protected, the MAP stores the instruction address in the *Violation address* register. The MAP then disables I/O interrupt requests, enters the supervisor mode, and directs the CPU to *jump indirect* to location 47_8. Any write to memory is inhibited.

Single cycle write protection works in the same way as normal write protection, but it can be enabled separately.

Auto-Increment/Decrement Protection

If auto-increment/decrement protection is enabled, any indirect reference to memory locations 20-37_8 will be considered a violation and will therefore trap. The system then stores the logical address of the instruction that caused the violation in the *Violation address* register, disables I/O interrupt requests, enters the supervisor mode, and directs the CPU to *jump indirect* to location 47_8.

Device Interrupt Processing

The MAP has been designed to allow for orderly processing of I/O interrupt requests by a supervisor program. When an I/O device requests an interrupt, the MAP sets the Program Map Inhibit bit in the MAP status word to 1. This immediately disables the translating of user addresses. That is, the Interrupt On flag is set to 0, the updated program counter is placed in physical memory location 0, and the CPU executes a *jump indirect* to physical memory location 1. A similar process occurs for stack overflow, normal trap instructions, and MAP violation traps.

To return control after an I/O interrupt, the supervisor can follow the method outlined previously (see *Setting Up for Translation*). The *Interrupt enable* instruction should be placed immediately before the JMP USERPC instruction.

FLOATING POINT UNIT

In addition to performing fixed point arithmetic, the NOVA 4 can perform floating point arithmetic if it is equipped with the floating point unit. This feature provides the capability to perform rapid and convenient arithmetic operations on numbers with a much larger range than would be feasible using the fixed point arithmetic instruction set. The precision with which these numbers can be manipulated exceeds the precision readily available with the fixed point instruction set.

Floating Point Unit Registers

There are three registers available to the programmer in the Floating Point Unit (FPU).

These are the Floating Point accumulator (FPAC), the Floating Point Status Register (FPSR), and the Temporary Buffer (TEMP). FPAC and TEMP are used for computations and FPSR is used to control and monitor the operation of the FPU.

FPAC and TEMP can both contain either single or double precision floating point numbers. FPSR is a 16-bit register containing bits that reflect the current status of FPAC and the FPU. The format of FPSR is as follows:

ANY	OVF	UNF	DVZ	MOF	GTZ	EQZ	LTZ						IOF		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

BITS	NAME	CONTENTS or FUNCTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator--while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator - while processing a floating point number, an exponent underflow occurred; the result is correct except that the exponent is 128 too large.
3	DVZ	Divide by Zero - while processing a floating point number, a zero divisor was detected; division was aborted and the operands remain unchanged.
4	MOF	Mantissa Overflow - during a .FSCL instruction, a left shift was required.
5	GTZ	Greater Than indicator; the operand in FPAC is positive and the mantissa is different from zero.
6	EQZ	Equal indicator; the operand in FPAC is equal to true zero. This bit examines only the mantissa and sign of FPAC.
7	LTZ	Less Than indicator; the operand in FPAC is less than zero.
8-12	---	Reserved for future use.
13	IOF	Interrupt off bit; the FPU will not interrupt the program for an exponent overflow, exponent underflow, or divide by zero.
14-15	---	Reserved for future use.

Interrupt Enable and Disable

To provide maximum flexibility, the FPU has an interrupt disable bit in the status register (bit 13), and is maskable via the *Mask out* instruction (bit 5 of the priority mask). If both these bits are set to 0, the FPU will signal an interrupt for exponent overflow, exponent underflow, or divide by zero. These conditions are represented by bits 1-3 in the status register. If either or both of the interrupt disable bits is set to 1, the FPU will not request an interrupt for any of the above conditions, but will set to one the representative bit in the status register and bit zero of the status register. These bits will remain set to 1 until cleared by the programmer. If running with interrupt disabled, it is the programmer's responsibility to test the status register periodically in order to detect errors in floating point processing.

NOTE: *The FPU returns 76₈ as the device code in response to the* INTA *instruction.*

INSTRUCTION SET

Because the FPU is considered an I/O device by the CPU, FPU instructions are really I/O instructions and take the I/O format. The device codes for the FPU are as follows:

MNEMONIC	DEVICE CODE	MEANING
FPU1	74_8	Floating Point - Single Precision
FPU2	75_8	Floating Point - Double Precision
FPU3	76_8	Floating Point Unit - used for status instructions and in diagnostic mode.

When processing a floating point instruction, the FPU assumes the following:

1. In instructions that refer to operands in memory, the accumulator specified by AC is assumed to contain the address of the first word of the storage that contains or will receive a floating point number. This area is either 2 or 4 words long, depending on the precision specified.

2. In instructions that refer to an operand coming from memory, the number is assumed to be in the format described under *Number Representation*. The number is assumed to be normalized.

3. In arithmetic instructions, it is assumed that a floating point number is already present in FPAC.

Load Floating Point Single

.FLDS   *ac*
DOBP   *ac*,FPU1

0	1	1	AC	1	0	0	1	1	1	1	1	1	0	0
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15
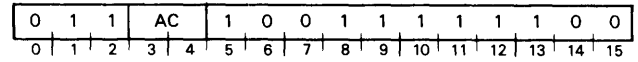
Moves two words out of memory into a specified FPAC.

Retrieves the single precision floating point number starting at the address contained in the specified AC and places it in FPAC. The low-order 32 bits of FPAC are set to 0. The operand in memory and the address in the specified AC remain unchanged.

Load Floating Point Double

.FLDD   *ac*
DOBP   *ac*,FPU2

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

Moves four words out of memory into FPAC.

Retrieves the double precision floating point number starting at the address contained in the specified AC and places it in FPAC. The operand in memory and the address in the specified AC remain unchanged.

V- 13

Store Floating Point Single

.FSRS   *ac*
DOBS   *ac*,FPU1

0	1	1	AC		1	0	0	0	1	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
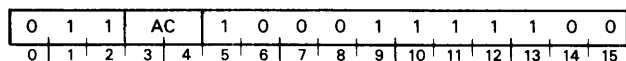
Stores the contents of FPAC into two sequential memory locations.

Places the floating point number contained in FPAC in memory beginning at the location addressed by the contents of the specified AC. Destroys the previous contents of the addressed memory location and leaves unchanged the contents of FPAC and the specified AC. Only the high-order 32 bits of FPAC are stored.

Store Floating Point Double

.FSRD   *ac*
DOBS   *ac*,FPU2

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

Stores the contents of a specified FPAC into four sequential memory locations.

Places the floating point number contained in FPAC in memory beginning at the location contained in the specified AC. Destroys the previous contents of the addressed memory location and leaves unchanged the contents of FPAC and the specified AC.

Add Single

.FAS   *ac*
DOA   *ac*,FPU1

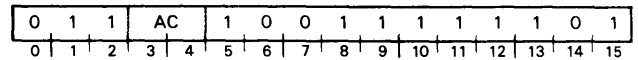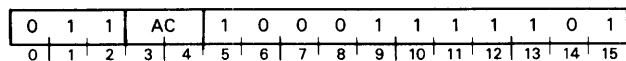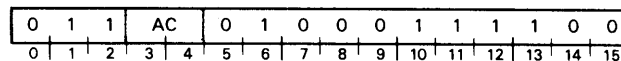| 0	1	1	AC		0	1	0	0	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Retrieves the single precision floating point number starting at the location addressed by the contents of the specified AC and adds it to the floating point number in FPAC, placing the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged. Sets the low-order 32 bits of FPAC to zero.

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. The last 8 bits shifted out are retained as hex *guard* digits.

If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 8 hex digits in single precision.

After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra. If the mantissa addition produces a carry out of the high-order bit, the mantissa in the intermediate result is shifted right one hex digit and the exponent is incremented by one. If this shift produces an exponent overflow, the OVF bit is set in the SR, and the instruction is terminated. In this case, the number in FPAC is correct except that the exponent is 128 too small.

If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in FPAC and the instruction terminates.

If the mantissa is non-zero, the intermediate result is normalized, and the number placed in FPAC. If the normalization results in an exponent underflow, the UNF bit is set in the SR and the instruction is terminated. The number in FPAC is correct except that the exponent is 128 too large.

Add Double

.FAD   *ac*
DOA   *ac*.FPU2
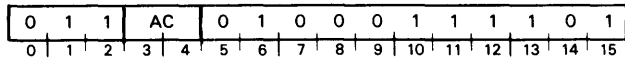
0	1	1	AC		0	1	0	0	0	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Retrieves the double precision floating point number starting at the location addressed by the contents of the specified AC and adds it to the floating point number in FPAC, placing the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged.

Floating point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. The bits shifted out of the right end of the mantissa are lost and do not take part in the addition.

If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 14 hex digits in double precision.

After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra. If the mantissa addition produces a carry out of the high-order bit, the mantissa in the intermediate result is shifted right one hex digit and the exponent is incremented by one. If this shift produces an exponent overflow, the OVF bit is set in the SR, and the instruction is terminated. In this case, the number in FPAC is correct except that the exponent is 128 too small.

If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in FPAC and the instruction terminates.

If the mantissa is non-zero, the intermediate result is normalized, and the number placed in FPAC. If the normalization results in an exponent underflow, the UNF bit is set in the SR and the instruction is terminated. The number in FPAC is correct except that the exponent is 128 too large.

Subtract Single

.FSS   *ac*
DOAS   *ac*,FPU1
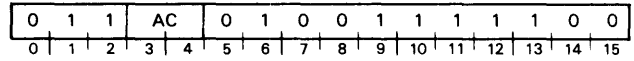
0	1	1	AC		0	1	0	0	1	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Retrieves the single precision floating point number starting at the location addressed by the contents of the specified AC and subtracts it from the floating point number in FPAC, placing the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged. Sets the low-order 32 bits of FPAC to zero.

Before the operation takes place, the sign bit of the operand fetched from memory is inverted. After the inversion, the operation is equivalent to addition.

Subtract Double

.FSD   *ac*
DOAS   *ac*,FPU2

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

Retrieves the double precision floating point number starting at the location addressed by the contents of the specified AC and subtracts it from the floating point number in FPAC, placing the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged.

Before the operation takes place, the sign bit of the operand fetched from memory is inverted. After the inversion, the operation is equivalent to addition.

Multiply Single

.FMS   *ac*
DOAP   *ac*,FPU1

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

Retrieves the single precision floating point number starting at the location addressed by the contents of the specified AC and multiplies it by the floating point number in FPAC, placing the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged. Ignores the low-order 32 bits of FPAC during the operation and sets them to zero in the result. The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding bit in the SR is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Multiply Double

.FMD   *ac*
DOAP   *ac*,FPU2

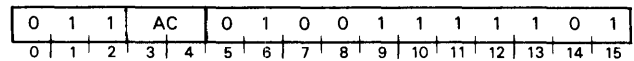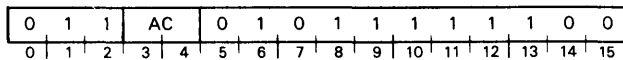| 0	1	1	AC	0	1	0	1	1	1	1	1	1	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Retrieves the double precision floating point number starting at the location addressed by the contents of the specified AC and multiplies it by the floating point number in FPAC, placing the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged.

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding bit in the SR is set to 1. The number is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Divide Single

.FDS   *ac*
DOA↰   *ac*,FPU1

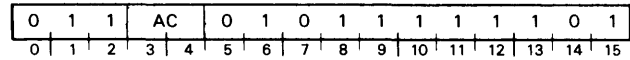| 0	1	1	AC	0	1	0	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the floating point number in FPAC by the single precision floating point number starting at the location addressed by the contents of the specified AC. Then places the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged. Ignores the low-order 32 bits of FPAC during the operation and sets them to zero in the result.

The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the SR and the instruction is terminated. The number in FPAC remains unchanged.

If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared and if the mantissa of the number in FPAC is greater than or equal to the mantissa of the source operand, the mantissa of the number in FPAC is shifted right one hex digit and the exponent of the number in FPAC is increased by one. Since all operands are assumed to be normalized, this guarantees that the mantissa of the number in FPAC will always be less than the mantissa of the source operand.

The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is normalized and placed in FPAC.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding bit in the SR is set to 1. The number in FPAC is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

NO! — "guard digits" used come from low-order 1st 8 bits.

Divide Double

.FDD   *ac*
DOAC   *ac*,FPU2

0	1	1	AC	0	1	0	1	0	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
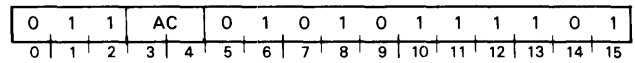
Divides the floating point number in FPAC by the double precision floating point number starting at the location addressed by the contents of the specified AC. Then places the normalized result in FPAC. Destroys the previous contents of FPAC, and leaves the contents of the source location and the address in the specified AC unchanged.
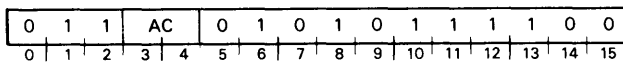
The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the SR and the instruction is terminated. The number in FPAC remains unchanged.

If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared and if the mantissa of the number in FPAC is greater than or equal to the mantissa of the source operand, the mantissa of the number in FPAC is shifted right one hex digit and the exponent of the number in FPAC is increased by one. Since all operands are assumed to be normalized, this guarantees that the mantissa of the number in FPAC will always be less than the mantissa of the source operand.

The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra. The result is normalized and placed in FPAC.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding bit in the SR is set to 1. The number in FPAC is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Temporary Buffer Instructions

The Temporary buffer, or TEMP, is a register in the FPU capable of holding a single or double precision floating point number. The following instructions use this register.

Move FPAC to TEMP

.FMFT
NIOP   FPU2

0	1	1	0	0	0	0	0	0	1	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
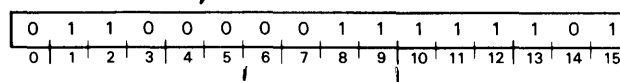
The double precision floating point number in FPAC is moved to the TEMP buffer. The number in FPAC remains unchanged.

If the previous instruction that referred to FPAC was a single precision instruction, then that instruction zeroed the low-order half of FPAC and the contents of FPAC can be handled as a double precision number.

Move Temp to FPAC

.FMTF
NIOC   FPU2

0	1	1	0	0	0	0	0	1	0	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
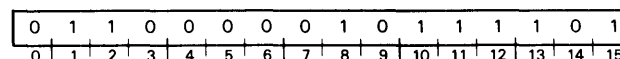
The double precision floating point number in the TEMP buffer is moved to FPAC. The number in the TEMP buffer remains unchanged.

If the previous instruction that referred to the TEMP buffer was a single precision instruction, then that instruction zeroed the low-order half of the TEMP buffer and the contents of the TEMP buffer can be handled as a double precision number.

Add TEMP to FPAC (Single)

.FATS
DOC   0,FPU1
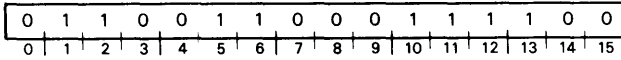
0	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Adds the floating point number in TEMP to the floating point number in FPAC and places the normalized result in FPAC. Leaves unchanged the number in TEMP. Only the high-order 32 bits of TEMP and FPAC participate in the operation.

This instruction is identical to the *Add single* instruction, except that the second operand comes from TEMP instead of memory.

Subtract TEMP from FPAC (Single)

.FSTS
DOCS   0,FPU1

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

Subtracts the floating point number in TEMP from the floating point number in FPAC and places the normalized result in FPAC. Leaves unchanged the number in TEMP. Only the high-order 32 bits of TEMP and FPAC participate in the operation.

This instruction is identical to the *Subtract single* instruction, except that the second operand comes from TEMP instead of memory.

Add TEMP to FPAC (Double)

.FATD
DOC   0,FPU2

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

Adds the floating point number in TEMP to the floating point number in FPAC and places the normalized result in FPAC. Leaves unchanged the number in TEMP.

This instruction is identical to the *Add double* instruction, except that the second operand comes from TEMP instead of memory.

Subtract TEMP from FPAC (Double)

.FSTD
DOCS   0,FPU2

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

Subtracts the floating point number in TEMP from the floating point number in FPAC and places the normalized result in FPAC. Leaves unchanged the number in TEMP.

This instruction is identical to the *Subtract double* instruction, except that the second operand comes from TEMP instead of memory.

Multiply FPAC by TEMP (Single)

.FMTS
DOCP   0,FPU1

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

Multiplies the floating point number in FPAC by the floating point number in TEMP and places the normalized result in FPAC. Leaves unchanged the number in TEMP. Only the high-order 32 bits of TEMP and FPAC participate in the operation.

This instruction is identical to the *Multiply single* instruction, except that the second operand comes from TEMP instead of memory.

Divide FPAC by TEMP (Single)

.FDTS
DOCC   0,FPU1
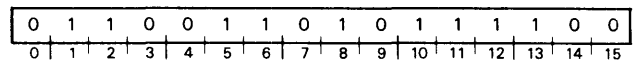
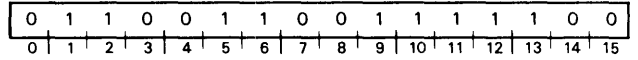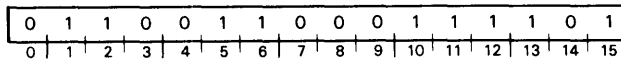| 0	1	1	0	0	1	1	0	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the floating point number in FPAC by the floating point number in TEMP and places the normalized result in FPAC. Leaves unchanged the number in TEMP. Only the high-order 32 bits of TEMP and FPAC participate in the operation.

This instruction is identical to the *Divide single* instruction, except that the second operand comes from TEMP instead of memory.

Multiply FPAC by TEMP (Double)

.FMTD
DOCP   0,FPU2
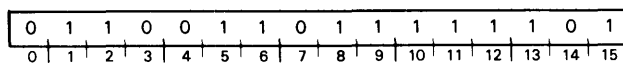
0	1	1	0	0	1	1	0	1	1	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Multiplies the floating point number in FPAC by the floating point number in TEMP and places the normalized result in FPAC. Leaves unchanged the number in TEMP.

This instruction is identical to the *Multiply double* instruction, except that the second operand comes from TEMP instead of memory.

Divide FPAC by TEMP (Double)

.FDTD
DOCC   0,FPU2
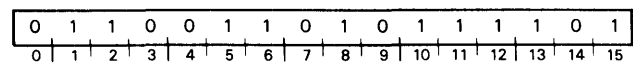
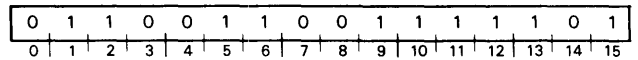| 0	1	1	0	0	1	1	0	1	0	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

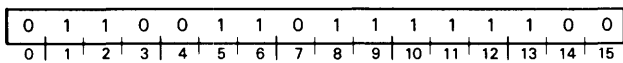Divides the floating point number in FPAC by the floating point number in TEMP and places the normalized result in FPAC. Leaves unchanged the number in TEMP.

This instruction is identical to the *Divide double* instruction, except that the second operand comes from TEMP instead of memory.

Shift and Logical Instructions

The FPU instructions are included to enable the programmer to convert numbers from integer representation to floating point representation and vice-versa. This section also contains instructions for logical operations and for working with the Status Register.

Absolute Value

.FABS
NIOP   FPU1

0	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
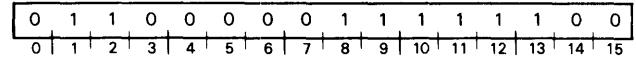
Forces the sign bit of FPAC to zero. Leaves bits 1-63 of FPAC unchanged.

Clear FPAC

.FCLR
NIOS   FPU1

0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
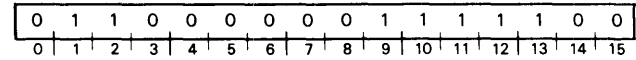
Forces all 64 bits of FPAC to zero. That is, the value of FPAC is forced to true zero.

FCLR

Load Exponent

.FLDX   *ac*
DOBC   *ac*,FPU2

0	1	1	AC	1	0	0	1	0	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
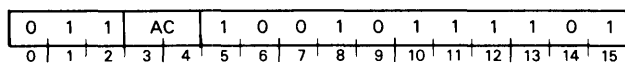
Places bits 1-7 of the specified AC in bits 1-7 of FPAC. Ignores bits 0 and 8-15 of the specified AC. Leaves unchanged bits 0 and 8-63 of FPAC and the entire contents of the specified AC.

> NOTE: *The instruction assumes that the exponent contained in bits 1-7 of AC is in* Excess 64 *representation.*

Normalize

.FNRM
NIOS   FPU2

0	1	1	0	0	0	0	0	0	1	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
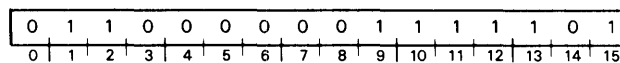
Normalizes the floating point number in FPAC. Sets a true zero in FPAC if all the bits of the mantissa are zero. Sets the UNF flag in the SR if an exponent underflow occurs. The number in FPAC is then correct, except that the exponent is 128 too large.

Negate

.FNEG
NIOC   FPU1

0	1	1	0	0	0	0	0	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
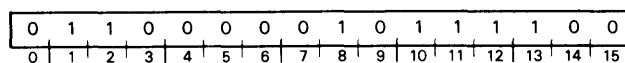
Inverts the sign bit of FPAC. Bits 1-63 of FPAC remain unchanged. If FPAC contains true zero, the sign bit remains unchanged.

Read High Word

.FHWD   *ac*
DIA   *ac*,FPU1

0	1	1	AC	0	0	1	0	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
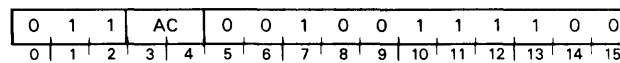
Places the high-order 16 bits of FPAC in the specified AC, destroys the previous contents of AC, and leaves unchanged the contents of FPAC.

Scale

.FSCL   *ac*
DOB   *ac*,FPU2

0	1	1	AC		1	0	0	0	0	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
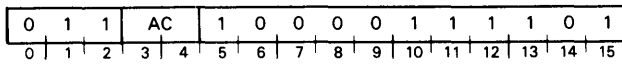
Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of the specified AC. Leaves the contents of AC unchanged.

Treats bits 1-7 of the specified AC as an exponent in *Excess 64* representation. Computes the difference between this exponent and the exponent in FPAC by subtracting the exponent in FPAC from the number contained in bits 1-7 of the specified AC. If the difference is zero, the instruction is terminated. If the difference is positive, the instruction shifts the mantissa contained in FPAC right that number of hex digits. If the difference is negative, the instruction shifts the mantissa contained in FPAC left that number of hex digits and the MOF bit in the SR is set to 1. After the shift, the contents of bits 1-7 of AC replace the exponent contained in FPAC. Bits shifted out of either end of the mantissa are lost. If the entire mantissa is shifted out of FPAC, the instruction sets FPAC to true zero.

(Not on original Nova)

Read Status

.FRST   *ac*
DIAC   *ac*,FPU

0	1	1	AC		0	0	1	1	0	1	1	1	1	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
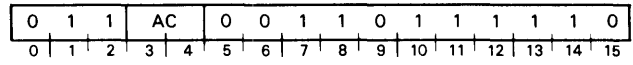
Places the contents of the 16-bit status register (SR) into the specified AC in the following format and sets bits 0-4 of the SR to zero.

ANY	OVF	UNF	DVZ	MOF	GTZ	EQZ	LTZ						IOF		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

| BITS	NAME	CONTENTS or FUNCTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator--while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator - while processing a floating point number, an exponent underflow occurred; the result is correct except that the exponent is 128 too large.
3	DVZ	Divide by Zero - while processing a floating point number, a zero divisor was detected; division was aborted and the operands remain unchanged.
4	MOF	Mantissa Overflow - during a .FSCL instruction, a left shift was required.
5	GTZ	Greater Than indicator; the operand in FPAC is positive and the mantissa is different from zero.
6	EQZ	Equal indicator; the operand in FPAC is equal to true zero. This bit examines only the mantissa and sign of FPAC.
7	LTZ	Less Than indicator; the operand in FPAC is less than zero.
8-12	---	Reserved for future use.
13	IOF	Interrupt off bit; the FPU will not interrupt the program for an exponent overflow, exponent underflow, or divide by zero.
14-15	---	Reserved for future use.

Write Status

.FWST *ac*
DOA *ac*,FPU

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

Places the contents of the spcified AC in the status register. Leaves unchanged the contents of the specified AC.

DOB   ac, FPU1

FPU Microcode Self test

NOVA Diag. mode

DIA   x, 74   FPAC[0]
DIB   x, 74   FPAC[1]
DIA   x, 75   FPAC[2]
DIB   x, 75   FPAC[3]

NIOP x, 74   "clock" FPU — busy set

.FWST

Chapter VI
VIRTUAL CONSOLE (VC)

VC is a program which permits you to perform control panel functions via the operator's console. Simple commands which you enter on a terminal keyboard allow you to examine and/or modify any processor register or memory location.

VC is supplied by Data General on the NOVA 4 as a set of read-only memory (ROM) units. They are not in the normal address space, so they are transparent to program operation.

The CPU may enter the VC program either upon power-up, or in response to the RESET switch or the console BREAK key. Certain conditions must be met by the state of the console keyswitch and the logic signal from the power supply which indicates that the contents of memory have been destroyed since power-down. The CPU may also execute a self-test routine to check the processor and the first 16k words of memory. The following table summarizes the CPU's response to various actions.

Keyswitch	Locked		Unlocked	
MEMORY STATE Action:	DATA OK	DATA LOST	DATA ' OK	DATA LOST
Power-up	Restart user program.	Self-test and enter VC	Enter VC (no self-test).	Self-test and enter VC
RESET switch	No function.	No function.	Enter VC (no self-test).	Self-test and enter VC
Console BREAK key	No function.	No function.	Enter VC (no self-test).	Enter VC (no self-test).

If it executes the self-test, VC types the letters *OK* on the console. Then it types an octal number, which is the value of the program counter when VC was entered. (On power-up, this number is 0.) VC then types a ! on the terminal. This is the *prompt*; it tells you that VC is in control and ready to accept a command.

Command Format

A VC command consists of a single character. Some commands must be preceded by an *argument* which is any octal number. Numbers that are used as memory addresses may be up to 17 bits long, to support the MAP. Numbers which are used as data are truncated to 16 bits.

If you wish to cancel the entire line that you have just entered, type a K. VC prints a ? followed by a new line, and also closes the current cell if it is open (described in detail below). The ? followed by a new line is also printed if you type a character which VC does not recognize, or in case VC detects a user error.

Cells

VC operates on *cells*. A cell is either a physical memory location, or an internal processor register (*internal cell*) such as an accumulator. In order to examine or modify any cell, you must *open* it. Opening a cell causes its contents to be printed, in octal, on your terminal.

To open an internal cell, use the command *n*A where *n* is one of the numbers listed in the table below.

INTERNAL CELLS

NUMBER OCTAL	CELL
0-3	The accumulators AC0 through AC3.
4	Return address: the contents of the program counter when VC was called.
5	Stack pointer.
6	Frame pointer.
7	bit 15: Interrupt enable flag: 0 = interrupts off. 1 = interrupts on.
10	MAP status word.
11	Switch register: the contents of this location are placed in an accumulator when a user program executes a READS instruction.
12	Bit 15: carry bit.

To open a memory location, use the / command. Typing *n*/ opens the location addressed by *n*. Typing / without an argument opens the cell addressed by the currently open cell (the *current cell*).

Modifying a Cell

Once you have opened a cell, you may change its contents by simply typing the number whose value is to be placed in the cell. Terminate the number by typing a carriage return or newline. If you type a newline, the next consecutive memory location or internal cell will be opened. This is convenient when you need to enter data into several consecutive cells.

Program Control

When VC is entered, it places the contents of the program counter into the 4A internal cell. Typing P causes VC to return to the location specifed by 4A.

You can also return to a program by typing *n*R. In this case, the CPU executes an IORST instruction, and resumes program execution at the location specifed by *n* (truncated to 15 bits).

Other commands which VC accepts are listed in the table below.

Command	Function
I	Initialize the system: execute an IORST, and clear the MAP.
*n*L	Perform a program load from device code *n*. (Bit 0 of *n* must be 1 for a data channel device.)
F	Perform a field service cassette load (for Data General use only).

APPENDIX   A

I/O DEVICE CODES AND DATA GENERAL MNEMONICS

OCTAL DEVICE CODE	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
00	----	--	Unused
01	MDV	--	Multiply/Divide
02	MAP	--	Memory Management Unit
03	MAP1		
05			
06	MCAT	12	Multiprocessor adapter transmitter
07	MCAR	12	Multiprocessor adapter receiver
10	TTI	14	TTY input
11	TTO	15	TTY output
12	PTR	11	Paper tape reader
13	PTP	13	Paper tape punch
14	RTC	13	Real-time clock
15	PLT	12	Incremental plotter
16	CDR	10	Card reader
17	LPT	12	Line printer
20	DSK	9	Fixed head disc
21	ADCV	8	A/D converter
22	MTA	10	Magnetic tape
23	DACV	--	D/A converter
24	DCM	0	Data communications multiplexor
25			
26			
27			
30	QTY	14	Asynchronous hardware multiplexor
30	SLA	14	Synchronous line adapter
31^2	IBM1	13	IBM 360/370 interface
32	IBM2		
33	DKP	7	Moving head disc
34	CAS	10	Cassette tape
34^2	MUX 8	11	Multiline asynchronous controller
35	CRC		
36	IPB	6	Interprocessor bus--half duplex
37	IVT	6	IPB watchdog timer
40	DPI	8	IPB full duplex input
41	DPO	8	IPB full duplex output
40^3	SCR	8	Synchronous communication receiver
41^4	SCT	8	Synchronous communication transmitter
42	DIO	7	Digital I/O
43	DIOT	6	Digital I/O timer

DG-01932

[2] Code returned by INTA

[3] Can be set up with any unused even device code equal to 40 or above

[4] Can be set up with any unused odd device code equal to 41 or above

I/O DEVICE CODES AND
DATA GENERAL MNEMONICS

OCTAL DEVICE CODE	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
44	MXM	12	Modem control for MX1/MX2
45			
46	MCAT1	12	Second multiprocessor transmitter
47	MCAR1	12	Second multiprocessor receiver
50	TTI1	14	Second TTY input
51	TTO1	15	Second TTY output
52	PTR1	11	Second paper tape reader
53	PTP1	13	Second paper tape punch
54	RTC1	13	Second real-time clock
55	PLT1	12	Second incremental plotter
56	CDR1	10	Second card reader
57	LPT1	12	Second line printer
60	DSK1	9	Second fixed head disc
61	ADCV1	8	Second A/D converter
62	MTA1	10	Second magnetic tape
63	DACV1	--	Second D/A converter
64			
65			
66			
67			
70	QTY1	14	Second asynchronous hardware multiplexor
70	SLA1	14	Second synchronous line adapter
71^2 72		13	Second IBM 360/370 interface
73	DKP1	7	Second moving head disc
74^2	FPU1		
75	FPU2	5	Floating point
76	FPU		
77	CPU	--	Central processor and console functions

[2]Code returned by INTA

[3]Can be set up with any unused even device code equal to 40 or above

[4]Can be set up with any unused odd device code equal to 41 or above

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.

	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

	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

APPENDIX C
ASCII CHARACTER CODES

Decimal	Octal	Hex	ASCII Character	Control Function	To Produce On TTY Mod 33, 35 Cntrl Shift Char			Even Parity 8-bit code
0	000	00	NUL	Null	✓	✓	P	00
1	001	01	SOH	Start of Heading	✓		A	81
2	002	02	STX	Start of Text	✓		B	82
3	003	03	ETX	End of Text	✓		C	03
4	004	04	EOT	End of Transmission	✓		D	84
5	005	05	ENQ	Enquiry	✓		E	05
6	006	06	ACK	Acknowledge	✓		F	06
7	007	07	BEL	Bell	✓		G	87
8	010	08	BS	Backspace	✓		H	88
9	011	09	HT	Horizontal Tab	✓		I	09
10	012	0A	NL	New Line			line feed	0A
					✓		J	0A
					✓		line feed	8A[1]
11	013	0B	VT	Vertical Tab	✓		K	8B
12	014	0C	FF	Form Feed	✓		L	0C
13	015	0D	RT	Return			return	8D
					✓		M	8D
					✓		return	0D[1]
14	016	0E	SO	Shift Out	✓		N	8E
15	017	0F	SI	Shift In	✓		O	0F
16	020	10	DLE	Data Link Escape	✓		P	90
17	021	11	DC1	Device Control 1	✓		Q	11
18	022	12	DC2	Device Control 2	✓		R	12
19	023	13	DC3	Device Control 3	✓		S	93
20	024	14	DC4	Device Control 4	✓		T	14
21	025	15	NAK	Negative Acknowledge	✓		U	95
22	026	16	SYN	Synchronous Idle	✓		V	96
23	027	17	ETB	End Transmission Block	✓		W	17
24	030	15	CAN	Cancel	✓		X	18
25	031	19	EM	End of Medium	✓		Y	99
26	032	1A	SUB	Substitute	✓		Z	9A
27	033	1B	ESC	Escape			esc	1B
					✓	✓	K	1B
28	034	1C	FS	File Separator	✓	✓	L	9C
29	035	1D	GS	Group Separator	✓	✓	M	1D
30	036	1E	RS	Record Separator	✓	✓	N	1E
31	037	1F	US	Unit Separator	✓	✓	O	9F
32	040	20	SP	Space			space	A0
33	041	21	!			✓	1	21
34	042	22	"			✓	2	22
35	043	23	#			✓	3	A3
36	044	24	$			✓	4	24
37	045	25	%			✓	5	A5
38	046	26	&			✓	6	A6
39	047	27	'			✓	7	27
40	050	28	(			✓	8	28
41	051	29	)			✓	9	A9

DG-01939

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

Decimal	Octal	Hex	ASCII Character	To Produce On TTY Mod 33, 35 Cntrl Shift Char	Even Parity 8-bit code
42	052	2A	*	✓ :	AA
43	053	2B	+	✓ ;	2B
44	054	2C	,	,	2C
45	055	2D	–	–	2D
46	056	2E	.	.	2E
47	057	2F	/	/	AF
48	060	30	0	0	30
49	061	31	1	1	B1
50	062	32	2	2	B2
51	063	33	3	3	33
52	064	34	4	4	B4
53	065	35	5	5	35
54	066	36	6	6	36
55	067	37	7	7	B7
56	070	38	8	8	B8
57	071	39	9	9	39
58	072	3A	:	:	3A
59	073	3B	;	;	BB
60	074	3C	<	✓ ,	36
61	075	3D	=	✓ –	BD
62	076	3E	>	✓ .	BE
63	077	3F	?	✓ /	3F
64	100	40	@	✓ P	C0
65	101	41	A	A	41
66	102	42	B	B	42
67	103	43	C	C	C3
68	104	44	D	D	44
69	105	45	E	E	C5
70	106	46	F	F	C6
71	107	47	G	G	47
72	110	48	H	H	48
73	111	49	I	I	C9
74	112	4A	J	J	CA
75	113	4B	K	K	4B
76	114	4C	L	L	CC
77	115	4D	M	M	4D
78	116	4E	N	N	4E
79	117	4F	O	O	CF
80	120	50	P	P	50
81	121	51	Q	Q	D1
82	122	52	R	R	D2
83	123	53	S	S	53
84	124	54	T	T	D4

DG-01939

APPENDIX C (Continued)
ASCII CHARACTER CODES

Decimal	Octal	Hex	ASCII Character	To Produce On TTY Mod 33, 35 Cntrl Shift Char	Even Parity 8-bit code
85	125	55	U	U	55
86	126	56	V	V	56
87	127	57	W	W	D7
88	130	58	X	X	D8
89	131	59	Y	Y	59
90	132	5A	Z	Z	5A
91	133	5B	[	✓ K	DB
92	134	5C	\	✓ L	5C
93	135	5D	]	✓ M	DD
94	136	5E	^	✓ N	DE
95	137	5F	—	✓ O	5F
96	140	60	`		60
97	141	61	a		E1
98	142	62	b		E2
99	143	63	c		63
100	144	64	d		E4
101	145	65	e		65
102	146	66	f		66
103	147	67	g		E7
104	150	68	h		E8
105	151	69	i		69
106	152	6A	j		6A
107	153	6B	k		EB
108	154	6C	l		6C
109	155	6D	m		ED
110	156	6E	n		EE
111	157	6F	o		6F
112	160	70	p		F0
113	161	71	q		71
114	162	72	r		72
115	163	73	s		F3
116	164	74	t		74
117	165	75	u		F5
118	166	76	v		F6
119	167	77	w		77
120	170	78	x		78
121	171	79	y		F9
122	172	7A	z		FA
123	173	7B	{		7B
124	174	7C				FC
125	175	7D	}		7D
126	176	7E	~		7E
127	177	7F	DEL	rubout	FF

DG-01939

C-3

DG OFFICES

SALES AND SERVICE OFFICES

Alabama: Birmingham
Arizona: Phoenix, Tucson
Arkansas: Little Rock
California: El Segundo, Fresno, Los Angeles, Oakland, Palo Alto, Pasadena, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Van Nuys
Colorado: Denver, Englewood
Connecticut: North Branford, Norwalk
Florida: Ft. Lauderdale, Orlando, Tampa
Georgia: Norcross
Idaho: Boise
Iowa: Cedar Rapids Bettendorf
Illinois: Arlington Heights, Champaign, Chicago, Peoria, Rockford, Schaumburg
Indiana: Indianapolis
Kentucky: Louisville
Louisiana: Baton Rouge, Metairie
Maine: Portland
Maryland: Baltimore
Massachusetts: Cambridge, Springfield, Wellesley, Worcester
Michigan: Grand Rapids, Southfield
Minnesota: Richfield
Missouri: Creve Coeur, Kansas City, St. Louis
Mississippi: Jackson
Montana: Billings
Nebraska: Omaha
Nevada: Las Vegas, Reno
New Hampshire: Bedford, Nashua
New Jersey: Cherry Hill, Somerset, Wayne
New Mexico: Albuquerque
New York: Albany, Buffalo, Lake Success, Latham, Melville, Newfield, New York, Rochester, Syracuse, White Plains
North Carolina: Charlotte, Greensboro, Greenville, Raleigh
Ohio: Brooklyn Heights, Cincinnatti, Columbus, Dayton
Oklahoma: Oklahoma City, Tulsa
Oregon: Lake Oswego, Portland
Pennsylvania: Blue Bell, Carnegie, Lancaster, Philidelphia, Pittsburgh
Rhode Island: Providence, Rumford
South Carolina: Columbia
Tennessee: Knoxville, Memphis, Nashville
Texas: Austin, Dallas, El Paso, Ft. Worth, Houston, San Antonio
Utah: Salt Lake City
Virginia: McLean, Norfolk, Richmond, Salem
Washington: Bellevue, Kirkland, Richland, Spokane
West Virginia: Charleston
Wisconsin: Brookfield, Madison, West Allis

INTERNATIONAL SUBSIDIARIES

Australia: Adelaide, Melbourne, New Castle, Sydney, Tasmania, Queensland, Victoria
Brazil: Sao Paulo
Canada: Calgary, Edmonton, Montreal, Ottawa, Quebec, Toronto, Vancouver, Winnipeg
France: Lille, Lyon, Nantes, Paris
Italy: Florence, Milan, Padua, Rome, Tourin
Japan: Tokyo
The Netherlands: Amsterdam, Rijswijk
New Zealand: Auckland, Wellington
Sweden: Gothenburg, Malmoe, Stockholm
Switzerland: Lausanne, Zurich
United Kingdom: Birmingham, Bristol, Chesire, Glasgow, Hounslow, London, Manchester
West Germany: Dusseldorf, Filderstadt, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Munich, Ratingen, Rodelheim, Stuttgart

REPRESENTATIVES & DISTRIBUTORS

Argentina: Buenos Aires
Bolivia: Novadata
Chile: Santiago
Columbia: Bogato
Costa Rica: San Jose
Ecuador: Quito
Egypt: Cairo
Finland: Helsinki
Guatemala: Guatemala City
Hong Kong: Hong Kong
India: Bombay
Indonesia: Jakarta
Israel: Givatayim
Korea: Seoul
Kuwait: Kuwait
Lebanon: Beirut
Malaysia: Kuala Lumpur
Mexico: Mexico City, Nuevo Leon
Morocco: Casablanca
Nicaragua: Managua
Nigeria: Ibadan, Lagos
Norway: Oslo
Paraguay: Asuncion
Peru: Lima
Philippine Islands: Manila
Portugal: Lisbon
Puerto Rico: Hato Rey
Saudi Arabia: Jaddah, Riyadh
Singapore: Singapore
South Africa: Capetown, Durban, Johannesburg, Pretoria
Spain: Barcelona, Bibao, Madrid, San Sebastian, Valencia
Taiwan: Taipei
Thailand: Bangkok
Turkey: Ankara
Uruguay: Montevideo
Venezuela: Maracaibo

ADMINISTRATION, MANUFACTURING RESEARCH AND DEVELOPMENT

Massachusetts: Cambridge, Framingham, Southboro, Westboro
Maine: Westbrook
New Hampshire: Portsmouth
California: Anaheim, Sunnyvale
North Carolina: Research Triangle Park, Johnston County
Texas: Austin

Hong Kong: Kowloon, Tai Po
Thailand: Bangkok

DG-04976

(Data General

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category
- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government
- ☐ Educational

5. Mode of Operation
- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other (Specify)		

6. Communications
- ☐ HASP
- ☐ RJE80
- ☐ RCX 70
- ☐ CAM
- ☐ XODIAC
- ☐ Other

Specify _____

7. Application Description

O _____
_____
_____
_____
_____
_____

3. Software
- ☐ AOS
- ☐ DOS
- ☐ MP/OS
- ☐ RDOS
- ☐ Other

Specify _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other
    Specify _____

4. Languages
- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

9. Users Group

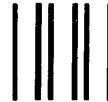Are you interested in joining a special interest or regional Data General Users Group?

O _____
_____

(Data General

Engineering Publications Comment Form

Please help us improve our future publications by answering the questions below. Use the space provided for your comments.

Title:_____

Document No. _____

Yes	No		
☐	☐	Is this manual easy to read?	○ You (can,cannot) find things easily. ○ Other: ○ Language (is,is not) appropriate. ○ Technical terms (are,are not) defined as needed.
		In what ways do you find this manual useful?	○ Learning to use the equipment ○ To instruct a class. ○ As a reference ○ Other: ○ As an introduction to the product
☐	☐	Do the illustrations help you?	○ Visuals (are,are not) well designed. ○ Labels and captions (are,are not) clear. ○ Other:
☐	☐	Does the manual tell you all you need to know? What additional information would you like?	
☐	☐	Is the information accurate? (If not, please specify with page number and paragraph.)	

Name: _____ Title: _____

Company: _____ Division: _____

Address: _____ City: _____
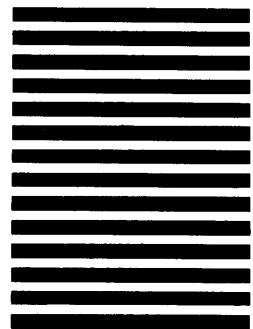
State: _____ Zip: _____ Telephone: _____ Date: _____

DG-05809

◖DataGeneral

Data General Corporation, Westboro, Massachusetts 01581

CUT ALONG DOTTED LINE

FOLD   FOLD

STAPLE   STAPLE

FOLD   FOLD

‖‖‖‖

BUSINESS REPLY MAIL
FIRST CLASS   PERMIT NO. 26   SOUTHBORO, MA.   01772

Postage will be paid by addressee:

◖� DataGeneral

ATTN: ENGINEERING PUBLICATIONS
4400 Computer Drive
Westboro, MA 01581