Programmer's Reference Series

ECLIPSE® S/250

Data General

Programmer's Reference Series

ECLIPSE® S/250

Data General

NOTICE

Data General Corporation (DGC) has prepared this manual for use by DGC personnel, licensees, and customers. The information contained herein is the property of DGC and shall not be reproduced in whole or in part without DGC's prior written approval.

Users are cautioned that DGC reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented, including, but not limited to typographical, arithmetic, or listing errors.

NOVA, INFOS, and **ECLIPSE** are registered trademarks of Data General Corporation, Westboro, Massachusetts. **DASHER**, **microNOVA**, and **ECLIPSE MV/8000** are trademarks of Data General Corporation, Westboro, Massachusetts.

Ordering No. 014-000611

Printed in the United States of America
Rev. 02, September, 1980

CONTENTS

Chapter 1
ECLIPSE® S/250 SYSTEM

The ECLIPSE S/250 system is Data General's most flexible scientific computer system. The advanced architectural features of this system provide exceptional configuration flexibility to accurately match your computing needs.

You can specify the ECLIPSE S/250 in configurations ranging from a small but powerful machine with standard ECLIPSE features, to a large specialized machine with an Array Processor, Satellite Processor Unit(s), and a Burst Multiplexor Channel.

In this chapter we discuss the features of the basic ECLIPSE S/250 and of the various options you can include in your system.

HIGHLIGHTS OF THE ECLIPSE S/250 SYSTEM

There are 4 systems making up the ECLIPSE S/250 which together are responsible for the processing power and throughput capability of this machine. They are:

- Main Storage system,
- I/O Management System,
- Main Processor,
- Packaging.

In this section we cover the highlights of these systems.

Main Storage

The ECLIPSE S/250 has a maximum memory capacity of 2 Mbyte when using Error Checking and Correcting (ERCC) semiconductor memory, with cycle times of 500 nanoseconds for read cycles, 600 nanoseconds for write cycles.

With core memory, the ECLIPSE S/250 has a maximum memory capacity of 512 Kbytes, with a cycle time of 800 nanoseconds.

Memory modules can be interleaved up to 8 ways; 4-way interleaving produces an effective cycle time as low as 300 nanoseconds for certain CPU instructions, and 200 nanoseconds for optional burst multiplexor I/O.

The ECLIPSE S/250 Memory Allocation and Protection unit protects individual user space within memory on a 2 Kbyte page basis. Protection modes include address validity, infinite defer, write, and I/O protection.

I/O Management System

The ECLIPSE S/250 has several systems for transferring information to and from the computer. Each method is appropriate for certain types of peripherals.

The optional Burst Multiplexor Channel (BMC) provides a direct communication path between main memory and high-performance peripherals such as DG/Disc Storage Subsystems and Fixed-Head DG/Discs. Maximum memory transfer rates are 10 Mbyte/second input, 6.7 Mbyte/second output. The BMC option can support up to 4 controllers transferring simultaneously.

The standard NOVA/ECLIPSE data channel provides I/O communication for medium-speed devices such as cartridge discs, magnetic tapes, data channel line printers, and synchronous communications. Maximum memory transfer rates are 2.5 Mbyte/second input, 1.7 Mbyte/second output.

Programmed I/O, with priority interrupt handling and vectoring capability for automatic dispatch to the correct interrupt handler, provides I/O communication for low-speed devices such as CRT

terminals, paper tape punches, and card readers.

Main Processor

The ECLIPSE S/250 main processor can execute the standard ECLIPSE instruction set, using a fast integer multiply/divide function implemented in firmware. It also executes the standard 56-instruction ECLIPSE floating point instruction set (FIS) implemented in firmware.

An optional Floating Point Processor (FPP) is available for extremely high-speed floating point operations. This two-board hardware option significantly reduces the execution times of floating point instructions. The FPP instructions are identical to the FIS instructions, and are capable of the same accuracy.

In addition to the standard 56-instruction ECLIPSE floating point instruction set, the FPP also performs the Extended Floating Point Functions. These instructions perform heavily-used scientific functions such as sine, cosine, square root, natural logarithms, exponentiation and polynomial evaluations.

The optional Character Instruction Set (CIS) simplifies handling of strings of characters or bytes. It is especially useful in communications and business applications, or any situation where strings of bytes must be moved, compared, or translated.

An optional User Control Store (UCS) provides facilities for creating your own microprogrammed subroutines. UCS can significantly reduce program running time if a frequently called compute-bound subroutine is converted to microcode.

There are two version of UCS: Fixed Control Store (FCS) and Writable Control Store (WCS). FCS provides hardware which permits you to insert PROMs containing microcode of your own design. WCS allows you to write your own microcode routines and load them into RAM memory, thus letting you change and debug the microcode routines. See *Learning to Microprogram Your ECLIPSE Computer (DGC No. 014-000098)* for more information about WCS.

Physical Design and Packaging

The ECLIPSE S/250 is packaged in a 19" wide rack cabinet, allowing vertical mounting of up to 34 boards in the mainframe. The ECLIPSE S/250 uses a 135-amp (+5V) power supply and a heavy-duty power distribution/fusing system. Multilayer printed circuit boards provide optimal power and signal distribution throughout. An additional 135-amp (+5V) power supply is available for systems with higher current requirements.

The ECLIPSE S/250 cabinet contains a heavy-duty blower system with over-temperature protection and integral cable troughs for routing cables to free-standing devices. External communications cables terminate cleanly at a rear patch panel, allowing quick rearrangment of communications lines.

The front console is relocatable for troubleshooting at the backplane and uses LED lamps for long life. It can address the entire physical address space, select individual maps, modify, examine or monitor individual locations, and freeze on address read or write.

Software Support

A wide variety of software support is available for the ECLIPSE S/250 system.

The Real-Time Disc Operating System (RDOS) supports both real-time and batch operations. It can efficiently manage up to 512 Kbytes of main memory in the ECLIPSE S/250 and supports foreground/background processing and multitasking.

The Advanced Operating System (AOS) uses adaptive resource management for efficient operation in multiuser environments. It can manage up to 1 Mbyte of main memory in the ECLIPSE S/250 and supports concurrent batch, timesharing, and interactive operations.

Many higher-level languages are also available, including Fortran IV, Fortran 5, Extended Basic, PL/I, DG/L (an ALGOL-derivative structured programming language), and Macro Assembler.

Integral Array Processor Option

The Integral Aray Processor (IAP) is designed to do signal and/or array processing in situations where large amounts of information must be processed in short amounts of time. IAP-augmented hardware and firmware make this possible by providing a comprehensive set of array processing instructions. This configuration enables a minicomputer system to provide performance formerly associated with much larger processors.

Major features of the IAP option include:

- the set of array/signal processing instructions that may be intermixed with standard instructions in a program. All floating point computations use Data General's standard single precision (32 bit) format, with one hex guard digit to insure high accuracy.
- 8K bytes of fast local memory that is part of the main memory address space. This memory may also be accessed as 16-, 32-, or 64-bit words. A 64-bit internal bus allows transfer of an entire real or complex number in 200 ns.
- Extensive software support including RDOS (Real-time Disk Operating System) and easy interfacing to FORTRAN and DG/L.

Architecture

The IAP is an integral part of the CPU, rather than a separate processor. This lowers the cost of software development as well as hardware, since no cross-assemblers or other specialized programs are needed.

The IAP, implemented on three 15-inch square printed circuit boards, is located in the CPU backplane along with the main memory. The IAP's *local memory* is part of the system memory's physical address space. This architecture eliminates extra data movements that would otherwise be required. Furthermore, the IAP has its own port into this local memory which can transfer data to and from the IAP with no interference to the system memory bus. Local memory is 4096 16-bit words, which reside in the highest 8 Kbytes of the physical address space. The ECLIPSE Memory Allocation and Protection unit (MAP) can map the local memory to any part of the user's logical address space. The IAP memory port is 64 bits wide, so that real or complex numbers can be read or written every 200 nanoseconds. This amounts to a maximum data transfer rate of 40 Mbytes per second.

Reliability

The IAP has a comprehensive set of maintainence instructions. These perform basic transfers along the IAP's internal data paths, enabling diagnostic programs to isolate any faults that may develop. This reduces the IAP's mean time to repair, thereby increasing the system's availibility.

Software Support

Although assembler language programming of the IAP is relatively simple, many users will wish to use the IAP to speed up their application programs written in higher level languages (notably FORTRAN). Data General supplies a library of subroutines that are compatible with existing Fortran 5 and DG/L compilers for the ECLIPSE, running under RDOS (Real-time Disk Operating System).

Each IAP instruction has a corresponding subroutine. Additional routines are provided that help the user with memory management and maintenance of parameter blocks. All of the IAP's features, such as error handling and use of the step registers, are made available to the high-level language programmer. For complete information on the IAP software support, refer to the *Array Processor Software User's Manual, DGC no. 093-000169.*

The Optional Satellite Processor

The Satellite Processor (SP) is a self-contained ECLIPSE CPU which works under the ECLIPSE S/250 (the host processor) to accomplish more efficient data management. Contained in the host itself, the SP is equipped with a MAP which allows it to access either its own 64K bytes of local memory or up to 64K bytes of host memory. The SP accesses host memory via the host data channel.

There are two models of SPs available. The 8660 SP comes equipped with the Character Instruction Set to make character manipulation and other similar tasks easy. The 8661 SP contains an integral array processor to efficiently process large amounts of data. Both the 8660 SP and the 8661 SP are user programmed.

The 8660 SP can be programmed to handle data coming from several I/O devices. It also can be programmed to handle much of the text editing chores of a system, thereby leaving the host free for efficient task management and other administrative jobs. A multiple SP system can be programmed to use the host as an intercommunications network between the SPs, with each SP handling its own group of I/O devices. SPs can perform any other communications function normally carried out by an ECLIPSE processor, including message concentration, network control, and real-time processing and control.

The 8661 SP can manipulate arrays and complex numbers efficiently and quickly. This means that the 8661 SP can be programmed to handle many scientific applications, such as signal processing, with ease.

The ECLIPSE S/250 can contain a maximum of 4 SPs. You can include both 8660 SPs and 8661 SPs in one system.

Highlights of the SP

In the following paragraphs we have highlighted some of the SP's architectural and design features that make possible the performance of the system.

- host-SP communications interface permitting full two-way communication between the processors, including a facility to boot the SP from the host
- cross-interrupt facility with the host processor, which permits both to operate independently until one requires attention
- an array processor (the 8661 SP only)
- both programmed and data channel I/O supported on its own I/O bus
- SP MAP enabling the SP to reference either its own local memory or the host's memory
- full ECLIPSE fixed point instruction set (the 8660 SP also recognizes the Character Instruction Set)
- priority interrupt handler taking care of up to 16 levels of interupts
- vectoring feature for automatic dispatch of interrupts to the correct interrupt handler
- microprogrammed instruction set
- push-down stacks for temporary information storage, with separately definable stack areas for

the operating system and the priority interrupt handler

The SP can be used with any Data General communications equipment. The SP's I/O bus will support all DG/CS communications equipment requiring installation in a communications chassis. The SP can support communications systems in the host main chassis, including:

- basic I/O controllers
- quad multiplexors
- synchronous line multiplexors
- universal line multiplexors
- DG/CS communications systems with external communications chassis

In external communications chassis, the SP will support DG/CS equipment including:

- asynchronous line multiplexors
- EIA interface modules
- 20ma current loop interface modules
- synchronous line multiplexors
- synchronous modem interfaces
- CRC generator

More information on DG/CS systems is available in the Technical Reference, Data General Communication Systems (DGC no. 014-000070).

Chapter II
CONCEPTS AND FACILITIES

The ECLIPSE S/250 combines all the standard ECLIPSE facilities, including:

the ECLIPSE standard instruction set,
the stack,
the I/O facilities,
the data channel,
the MAP,

with a variety of extremely powerful optional features such as:

Burst Multiplexor Channel,
Integral Array Processor,
Satellite Processor,
Writable Control Store,
Floating Point Functions,
Character Instructions.

The result is a machine with the configuration flexibility to match the needs of a wide variety of users.

In this chapter we describe the facilities which are standard on all ECLIPSE S/250s, and the assembly-language instructions which control these facilities. In the next chapter, we describe the optional facilities and their instructions.

You can find complete descriptions of all the ECLIPSE S/250 assembly-language instructions, other than I/O instructions, in Chapter IV. Chapter V contains complete descriptions of all the I/O instructions.

ADDRESSING CONVENTIONS

The various methods of addressing memory locations in the ECLIPSE S/250 give you considerable flexibility when storing and retrieving data, or transferring control to a different procedure.

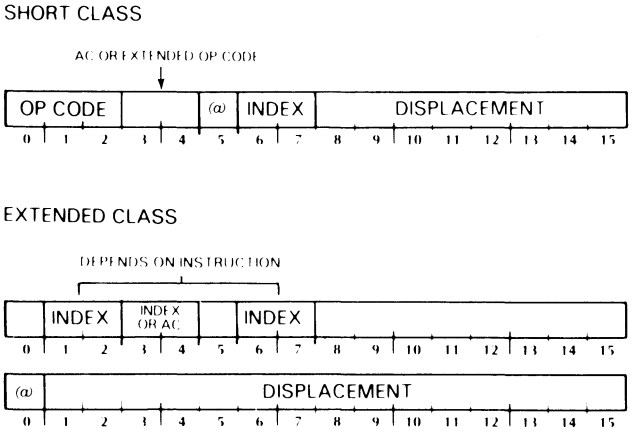
Each addressed location in main memory consists of a 16-bit word. The first word in memory has the address 0, the next has the address 1, the next 2, and so forth.

In this manual, we speak of a user's *address space* of 2^{15} words. This is a reference to the *logical* address space; the address space the user normally sees, which can be addressed by a 15-bit address. The maximum amount of logical address space available to the programmer is 32,768 words. (The *physical* address space - corresponding to the total amount of main memory in the computer - may be much larger.) Within a logical address space, the next sequential memory location after location 77777_8 is location 0.

The MAP controls the relationship between a logical address space and the physical address space by translating logical addresses to physical addresses. When the MAP is enabled, it intercepts each memory reference and translates the 15-bit logical address into a 20-bit physical address. Unless the MAP itself is being programmed, the translation process is invisible to the programmer.

Word Addressing Definitions

The following definitions are useful for understanding word addressing in the ECLIPSE S/250:



Addressing Modes - Methods of addressing using a displacement from some reference point to find the desired address. There are three different modes, each using different reference points.

Indirect Addressing - A method of addressing which uses the first address found as a pointer to another address which, in turn, may be used as a pointer to yet another address, etc. A series of indirect addresses is called an *indirection chain.*

Index Bits - Bits in the instruction which control the addressing mode used when executing this instruction.

Indirect Bit - A bit in the instruction or address which controls the indirection chain at each step of the addressing process.

Displacement Bits - Bits in the instruction which control the displacement distance, in memory locations, between some reference point (determined by the mode) and the desired address.

Effective Address Calculation - Logical process of converting the index, indirect, and displacement bits into an address to be used by the instruction.

Intermediate Address - The address obtained by the effective address calculation before testing for indirection.

Lower Page Zero - Locations $0\text{-}377_8$ in memory.

When the index bits are 00, the displacement is considered an unsigned integer. When the index bits are 01, 10, or 11, the displacement is considered a signed integer. Following is a table for the range of the displacement field under various conditions.

INDEX BITS	RANGE OF DISPLACEMENT FIELD	
	SHORT CLASS	EXTENDED CLASS
00	0 to 377_8 or 0 to 255_{10}	0 to 77777_8 or 0 to $32{,}767_{10}$
01	-200_8 to 177_8	-40000_8 to 37777_8
10	or	or
11	-128 to $+127_{10}$	$-16{,}384$ to $+16{,}383_{10}$

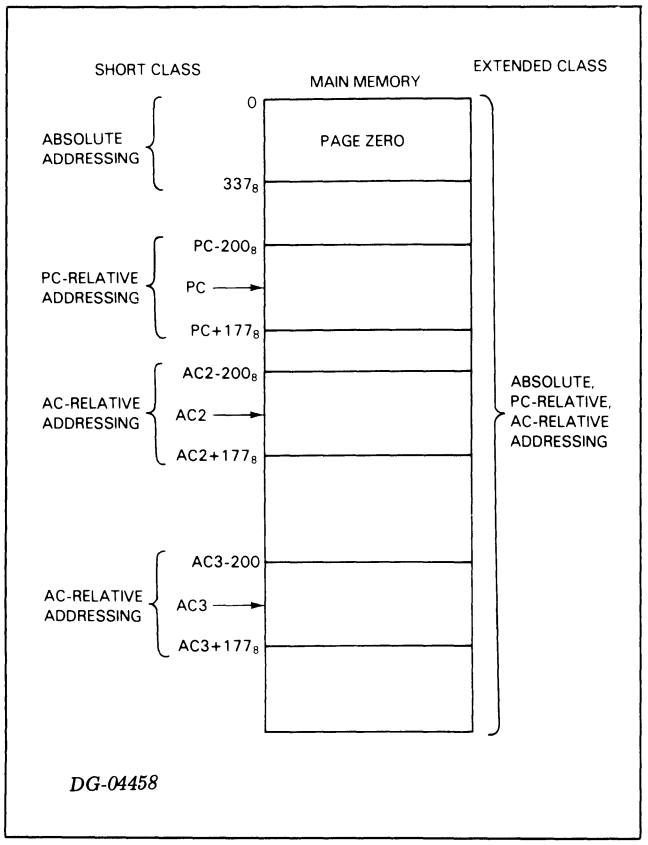
Addressing Modes

Word addressing in the ECLIPSE S/250 can be done in the following modes:

- absolute addressing;
- P.C. (program counter) relative addressing;
- accumulator relative addressing.

In addition, direct or indirect addressing can be used in any of these modes. By choosing the proper mode at the appropriate time, you can obtain access to any address in your logical address space.

The figure below illustrates the three addressing modes.



Absolute Addressing Mode - In absolute addressing mode, the intermediate address is set equal to the unmodified displacement. As a result, the short class of instructions specify locations in the range $0\text{-}377_8$ in the absolute mode (short class instructions are restricted to 8 bits in the displacement).

Lower page zero thus becomes very important because any memory-reference instruction can address this area. You can use it as a common storage area for items that you frequently reference throughout a program. Note, however, that we reserve some of these locations for special purposes.

Extended class instructions can reference any logical memory address using the absolute addressing mode.

P.C. Relative Addressing Mode - In P.C. relative addressing mode, the intermediate address is found by adding the displacement to the address of the word containing the displacement.

Accumulator Relative Addressing Mode - In accumulator relative addressing mode, the intermediate address is found by adding the displacement to the contents of bits 1-15 of the accumulator indicated by the index bits (you may use either AC2 or AC3).

Direct and Indirect Addressing

Direct addressing uses the intermediate address without modification.

Indirect addressing uses the intermediate address as a pointer to the next address. If bit 0 of the next address is 1, this address is used as a *pointer* which points to another address. The indirection chain is continued until an address is found with bit 0 equal to 0. This address is then used as the address of the data.

Any number of indirection levels is permitted in the ECLIPSE S/250, but indirect protection is available which can limit indirections to 15 levels (see the MAP section).

Auto-Incrementing and Auto-Decrementing

With the exception of memory references made *i)* within the CPU's Map A or Map B logical address spaces and *ii)* while demand paging is enabled, certain reserved locations within lower page zero perform auto indexing. Auto indexing proceeds as follows: If the intermediate address of a short class instruction is in the range $20\text{-}27_8$, and the indirect bit is 1, the contents of the addressed location are incremented by one, and the addressing chain continues using the *incremented* value of the addressed location.

If the intermediate address of a short class instruction is in the range $30\text{-}37_8$, and the indirect bit is 1, the contents of the addressed location are decremented by one, and the addressing chain continues using the *decremented* value of the addressed location. Within CPU Map A and Map B logical address spaces, and when demand paging is enabled, indirection chains pass through locations $20_8\text{-}37_8$ normally - that is, without altering the auto index location(s).

> **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 intermediate address in the chain, rather than the effective address.*

You can find a flow diagram of the addressing process in Appendix F.

BIT MANIPULATION

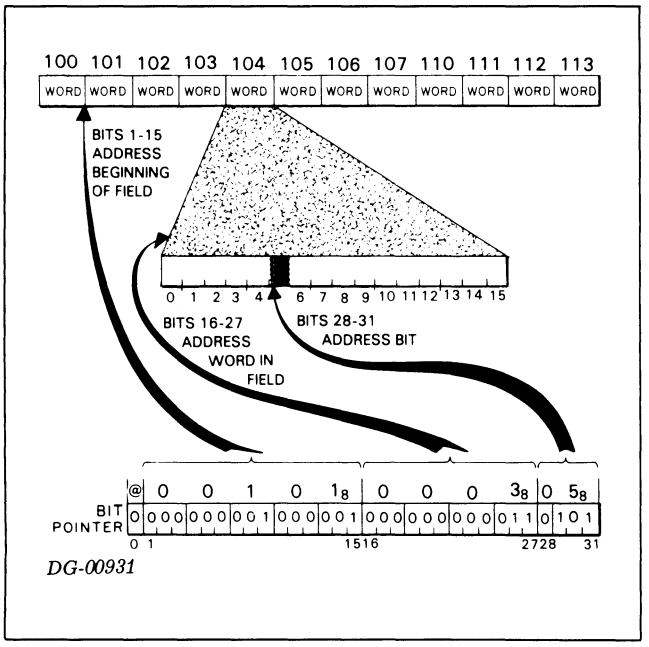
Bit Addressing

We use a 32-bit (2-word) *bit pointer* to address individual bits in memory. Bit 0 of the bit pointer is the indirect bit. If this bit is 1, the indirection chain (using bits 1-15 for the address each time) will be followed until a word is found with bit 0 set to 0. Bits 1-15 of this word become bits 1-15 of the bit pointer, and bits 0-15 of the next word become bits 16-31 of the bit pointer.

We form the address of the desired bit as follows:

The address formed by the unsigned number contained in bits 1-15 of the bit pointer (the *base* address) is added to the number formed by the 12-bit unsigned number contained in bits 16-27 (the *offset*). The resulting address points to the word containing the desired bit. Bits 28-31 of the bit pointer contain a 4-bit unsigned number which is the number of the desired bit in the addressed word.

Below is a diagram of the bit-addressing process.



Bit Instructions

The ECLIPSE S/250 instructions which manipulate bits:

- Locate a bit in memory and set it to 0 or 1;
- Test a bit, skipping the next word if the specified condition is true;
- Add a number to the contents of one accumulator based on the number of ones or high-order zeros found in the other accumulator.

Some of the bit instructions use a bit pointer to locate a bit in memory. The others only affect bits within the specified accumulators.

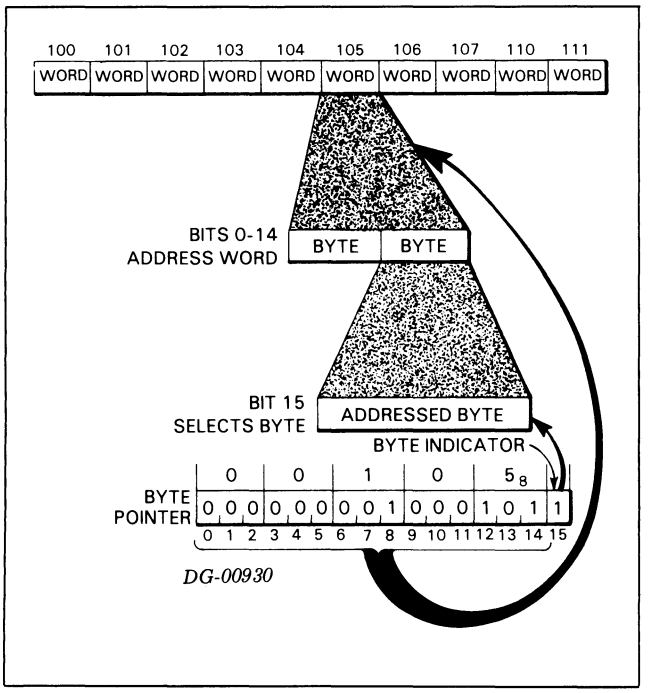
BIT MANIPULATION INSTRUCTIONS

Mnem	Name	Function
BTO	Set Bit To One	Sets the bit addressed by the bit pointer to 1.
BTZ	Set Bit To Zero	Sets the bit addressed by the bit pointer to 0.
COB	Count Bits	Counts the number of ones in one accumulator and adds that number to the second accumulator.
LOB	Locate Lead Bit	Counts the number of high-order zeros in one accumulator and adds that number to the second accumulator.
LRB	Locate And Reset Lead Bit	Performs a *Locate Lead Bit* instruction and sets the lead bit to 0.
SNB	Skip On Non-Zero Bit	Skips the next sequential word if the bit addressed by the bit pointer is 1.
SZB	Skip On Zero Bit	Skips the next sequential word if the bit addressed by the bit pointer is 0.
SZBO	Skip On Zero Bit And Set To One	Sets the bit addressed by the bit pointer to 1 and skips the next sequential word if the bit were originally 0.

BYTE MANIPULATION

Byte Format

We represent bytes as 8-bit unsigned binary integers. A byte in memory is selected by a 16-bit *byte pointer*. Bits 0-14 of the byte pointer contain the memory address of a 2-byte word. Bit 15 (the *byte indicator*) indicates which byte of the addressed location will be used. If bit 15 is 0, the high-order byte (bits 0-7) will be used. If bit 15 is 1, the low-order byte (bits 8-15) will be used. See the figure below.



Byte Instructions

The byte instructions are shown in the table below. Note that when an instruction moves a byte 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.

The two extended instructions (**ELDB** and **ESTB**) use a byte pointer contained in the instruction coding to reference bytes. The two short class instructions (**LDB** and **STB**) use an accumulator to hold the byte pointer.

Byte Instructions

Mnem	Name	Function
LDB ELDB	Load Byte	Places a byte of information into an accumulator.
STB ESTB	Store Byte	Stores the right byte of an accumulator into a byte of memory.

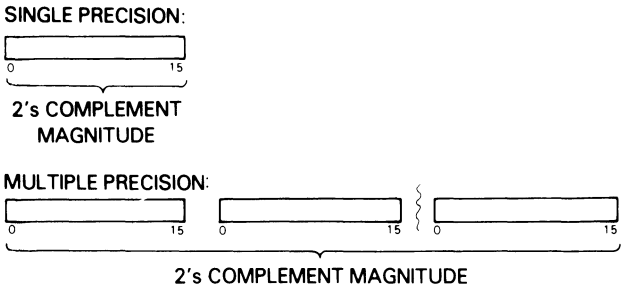
NUMBER MANIPULATION

Integer Format

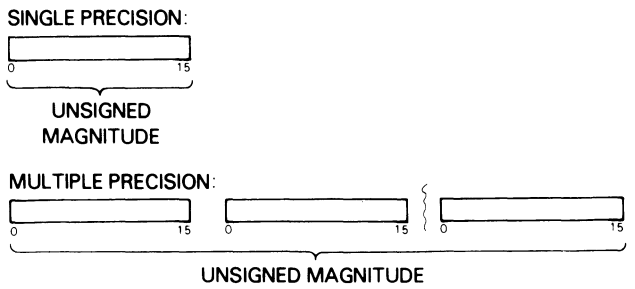
We represent a signed integer by a two's-complement number in one or more 16-bit words. The sign of the number is positive if bit 0 of the first word is 0 and negative if that bit is 1.

We represent an unsigned integer by using all the bits of one or more 16-bit words to represent the magnitude.

SIGNED INTEGERS



UNSIGNED INTEGERS



Single precision integers are one word (16 bits) long, and multiple precision integers are two or more words long. As an example, the table below shows the possible range of single and double precision numbers represented by this format:

	Single Precision	Double Precision
Unsigned	0 to 65,535	0 to 4,294,967,295
Signed	-32,768 to +32,767	-2,147,483,648 to +2,147,483,647

In addition, there is a value called *carry*. A change in the value of carry indicates an overflow during fixed point arithmetic operations.

Fixed Point Arithmetic Instructions

There are 26 ECLIPSE S/250 instructions which perform fixed point arithmetic. These instructions:

- Perform binary arithmetic on operands in accumulators;
- Load data from memory to an accumulator;
- Store data from an accumulator into memory;

All of the fixed point arithmetic instructions are shown in the following table. Some of the instructions appear in both a short form and a long form (the long form is usually indicated by the prefix *E* in the mnemonic). Most of these are instructions that move data. Short form instructions are 16 bits in length and can directly specify a memory address from 0 to 377_8, or can directly specify a small area in memory surrounding the present value of the program counter or an accumulator. Long form instructions are 32 bits in length; they can directly specify any address from 77777_8.

ADI and **ADDI** are also short and long forms, respectively, of the same instruction. The short form can only add a 2-bit immediate in the range 1-4, while the long form can add a 16-bit immediate in the range -32,768 to +32,767.

FIXED POINT INSTRUCTIONS

Mnem	Name	Function
ADC	Add Complement	Adds the logical complement (1's complement) of the contents of one accumulator to the contents of another accumulator.
ADD	Add	Adds contents of two accumulators.
ADDI	Extended Add Immediate	Adds a signed integer in the range -32,768 to +32,767 to the contents of an accumulator.
ADI	Add Immediate	Adds an unsigned integer in the range 1-4 to the contents of an accumulator.
DIV	Unsigned Divide	Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator.
DIVS	Signed Divide	Divides the signed 32-bit integer in two accumulators by the signed contents of a third accumulator.
DIVX	Sign Extend And Divide	Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.
DSZ EDSZ	Decrement And Skip If Zero	Decrements the addressed word, then skips if the decremented value is zero.
HLV	Halve	Divides the unsigned contents of an accumulator by 2.
INC	Increment	Increments the contents of an accumulator.
ISZ EISZ	Increment And Skip If Zero	Increments the addressed word, then skips if the incremented value is zero.
LDA, ELDA	Load Accumulator	Loads data from memory to an accumulator.
LEF, ELEF	Load Effective Address	Places an effective address in an accumulator.
MOV	Move	Moves the contents of an accumulator through the Arithmetic Logic Unit (ALU).
MUL	Unsigned Multiply	Multiplies the unsigned contents of two accumulators and adds the results to the unsigned contents of a third accumulator.
MULS	Signed Multiply	Multiplies the signed contents of two accumulators and adds the results to the signed contents of a third accumulator.
NEG	Negate	Forms the two's complement of the contents of an accumulator.
SBI	Subtract Immediate	Subtracts an unsigned integer in the range 1-4 from the contents of an accumulator.
STA, ESTA	Store Accumulator	Stores data in memory from an accumulator.
SUB	Subtract	Subtracts contents of one accumulator from another.
XCH	Exchange Accumulators	Exchanges the contents of two accumulators.

Decimal Format

Unsigned decimal numbers are handled one decimal digit at a time. Each decimal digit is represented by bits 12-15 of a 16-bit word. Only the values $0\text{-}9_{16}$ are used; carry is used for a decimal carry or borrow.

Decimal Arithmetic Instructions

There are 2 instructions in the ECLIPSE S/250 which perform operations on decimal data. These instructions add and subtract decimal integers.

DECIMAL ARITHMETIC INSTRUCTIONS

Mnem	Name	Function
DAD	Decimal Add	Adds together the decimal digits found in bits 12-15 of two accumulators.
DSB	Decimal Subtract	Subtracts the decimal digit in bits 12-15 of one accumulator from the decimal digit in bits 12-15 of another acumulator.

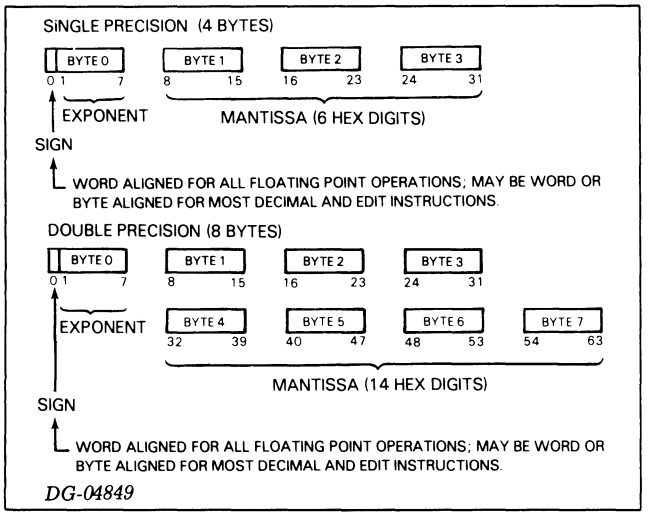
Floating Point Format

Word for word, floating point format provides a much larger range than integer format, at the expense of some precision. It also provides the ability to operate on fractions. The maximum range of floating point format is 5.4×10^{-79} to 7.2×10^{75}, the equivalent to a 16-word multiple precision integer. In addition, floating point operations are executed faster than most multiple precision integer operations.

We represent a floating point value using a 4-byte-wide (for single precision) or an 8-byte-wide (for double precision) number. The 4- or 8-byte aggregate contains 3 fields:

- a fractional part called the mantissa, which, at the end of all floating point mathematics operations, is always adjusted to be greater than or equal to 1/16 and less than 1 (i.e., *normalized*);
- an exponent, which is adjusted to maintain the correct value of the number;
- a sign.

Operations on numbers in memory employing the floating point arithmetic instructions require that the number be *word aligned*, so that bit 0 of the first byte of the number is bit 0 of first word of a 2-word or 4-word area in memory. Certain operations on numbers in memory employing decimal or edit instructions allow the number to be either word aligned or *byte aligned*. Byte alignment means that bit 0 of the first byte of the number is either bit 0 or bit 8 of any word in memory.



The magnitude of a floating point number is defined to be:

$$\text{MANTISSA} \times 16^{(\text{TRUE VALUE OF THE EXPONENT})}$$

We represent zero in floating point by a number with all bits zero, known as *true zero*. When a calculation results in a zero mantissa, the number is automatically converted to a true zero.

Sign

Bit 0 of the first byte is the sign bit. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative.

Exponent

The right-most 7 bits of the first byte contain the exponent. We use *excess 64* representation. For both positive and negative exponents, the value is 64 greater than the true value of the exponent. The following table illustrates this:

EXPONENT FIELD	TRUE VALUE of EXPONENT
0	-64
64	0
127	+63

Mantissa

Bytes 1-3 (single precision) or bytes 1-7 (double precision) contain the mantissa. By definition, the binary point lies *between* byte 0 and byte 1 of a floating point number. In order to keep the mantissa in the range of 1/16 to 1, the results of each floating point calculation are *normalized*. A mantissa is normalized by shifting it left one hex digit (4 bits) at a time, until the high-order four bits (the left-most four bits of byte 1) represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one.

Floating Point Arithmetic

The ECLIPSE S/250 floating point instructions assume normalized input numbers. Results are undefined for unnormalized input.

Floating Point Registers

There are five registers available to the programmer in the floating point processor. These are the four floating point accumulators (FPAC's) and the Floating Point Status Register (FPSR). The FPAC's are numbered 0-3 and are called FAC0, FAC1, FAC2, and FAC3. The FPSR is a 32-bit register that contains information about the present status of the floating point processor. The format of the FPSR is given at right.

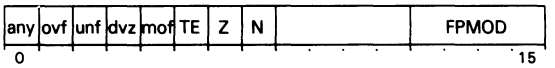
Guard Digit

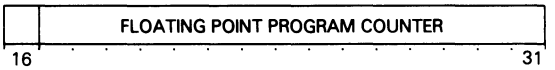
In order to increase accuracy, a 4-bit (1 hex digit) *guard digit* is used during floating point arithmetic operations. This guard digit accepts and holds up to 4 bits shifted out (to the right) of the mantissa, and is used in all single precision and double precision operations until the completion of each instruction. The guard digit is truncated before the data is stored at the end of the instruction process.

Floating Point Fault Conditions

After every floating point operation, the floating point status register is checked for possible fault conditions. Four types of floating point fault conditions can be detected:

- overflow
- underflow
- divide by zero
- mantissa overflow





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 FSCAL instruction, a significant bit was shifted out of the high order end of the mantissa; this bit is also set during a FIX instruction if the result does not fit into the destination location.
5	TE	Trap Enable - If this bit is 1, setting any of bits 1-4 will result in a floating point fault.
6	Z	Zero bit - The result of the last floating point operation was zero.
7	N	Negative bit--The result of the last floating point operation was less than zero.
8-11*	---	Reserved for future use.
12-15	FPMOD	Indicates computer series supporting the floating point instruction set. 0000 S/200, C/300, S/230, C/330 0001 S/130, S/250 standard FP 0010 M/600, C/350, S/250 optional FP 0011 Reserved for future use. 0100 Reserved for future use. 0101 8660 SP, 8661 SP 0110 C/150, S/250 standard EAU 0111- Reserved for 1111 future use.
16	---	Reserved for future use.
17-31	FPPC	Floating Point Program Counter - This is the logical address of the last floating point instruction executed. In the event of a floating point fault, this is the address of the floating point instruction that caused the fault.

**These bits are used as internal flags by the firmware; Preserve them when saving the state of the FPSR.*

Floating Point Trap

If the program has set bit 5 of the floating point status register to 1, a floating point fault condition will initiate a floating point trap. Immediately before the next floating point instruction is executed, a return block is pushed onto the stack and the program counter jumps indirect via location 45_8. Location 45_8 should contain the address of the floating point fault handler. The return block pushed has the following format:

WORD	DESCRIPTION
0	AC0
1	AC1
2	AC2
3	AC3
4	Bit 0: Carry; Bit 1-15: return address

NOTE: *The return address is not the address of the floating point instruction that caused the fault, nor is it (necessarily) the address of the instruction following the instruction that caused the fault. It is the address of the* floating point *instruction following the instruction that caused the fault.*

If the instruction following the instruction that caused the fault is a Push Floating Point State *or a* Pop Floating Point State, *the fault will not occur immediately. It will occur when the system returns to the same user environment and is about to execute a floating point instruction other than a* Push Floating Point State *or a* Pop Floating Point State *. In this way, the fault will only occur within the user environment which caused it.*

The floating point instructions are shown in the following table. Note that several instructions have two forms; one ending in *S* and one ending in *D* . The first form uses single-precision floating point format, while the second form uses double-precision floating point format. The function of the two forms is otherwise identical.

FLOATING POINT INSTRUCTIONS

Mnem	Name	Function
FAB	Absolute Value	Sets the sign bit of an FPAC to 0.
FAMS, FAMD	Add (memory to FPAC)	Adds the floating point number in memory to the floating point number in an FPAC.
FAS, FAD	Add (FPAC to FPAC)	Adds the floating point number in one FPAC to the floating point number in another FPAC.
FCLE	Clear Errors	Sets bits 0-4 of the FPSR TO 0.
FCMP	Compare Floating Point	Compares two floating point numbers and sets the Z and N flags accordingly.
FDMS, FDMD	Divide (FPAC by memory)	Divides the floating point number in an FPAC by a floating point number in memory.
FDS, FDD	Divide (FPAC by FPAC)	Divides the floating point number in one FPAC by the floating point number in another FPAC.
FEXP	Load Exponent	Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC.
FFAS	Fix To AC	Converts the integer portion of a floating point number to a signed two's complement integer and places the result in an accumulator.
FFMD	Fix To Memory	Converts the integer portion of a floating point number to double-precision integer format and stores the result in two memory locations.
FHLV	Halve	Divides the floating point number in FPAC by 2.
FINT	Integerize	Sets the fractional portion of the floating point number in the specified FPAC to zero and normalizes the result.
FLAS	Float From AC	Converts a signed two's complement number in an accumulator to a single precision floating point number.
FLDS, FLDD	Load Floating Point	Copies a floating point number from memory to a specified FPAC.
FLMD	Float From Memory	Converts the contents of two memory locations in integer format to floating point format and places the result in a specified FPAC.
FLST	Load Floating Point Status	Copies the contents of two specified memory locations to the FPSR.
FMMS, FMMD	Multiply (memory by FPAC)	Multiplies the floating point number in memory by the floating point number in an FPAC.

FLOATING POINT INSTRUCTIONS (Continued)

Mnem	Name	Function
FMOV	Move Floating Point	Moves the contents of one FPAC to another FPAC.
FMS, FMD	Multiply (FPAC by FPAC)	Multiplies the floating point number in one FPAC by the floating point number in another FPAC.
FNEG	Negate	Inverts the sign bit of the FPAC.
FNOM	Normalize	Normalizes the floating point number in FPAC.
FNS	No Skip	No operation.
FPOP	Pop Floating Point State	Pops an 18-word floating point block off the user stack and alters the state of the floating point unit.
FPSH	Push Floating Point State	Pushes an 18-word floating point block onto the user stack.
FRH	Read High Word	Places the high-order 16 bits of an FPAC in AC0.
FSA	Skip Always	Skips the next sequential word.
FSCAL	Scale	Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0.
FSEQ	Skip On Zero	Skips the next sequential word if the Z flag of the FPSR is 1.
FSGE	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if the N flag of the FPSR is 0.
FSGT	Skip On Greater Than Zero	Skips the next sequential word if both the Z and N flags of the FPSR are 0.
FSLE	Skip On Less Than Or Equal To Zero	Skips the next sequential word if either the Z flag or the N flag of the FPSR is 1.
FSLT	Skip On Less Than Zero	Skips the next sequential word if the N flag of the FPSR IS 1.
FSMS, FSMD	Subtract (memory from FPAC)	Subtracts the floating point number in memory from the floating point number in an FPAC.
FSND	Skip On No Zero Divide	Skips the next sequential word if the divide by zero (DVZ) flag of the FPSR is 0.

FLOATING POINT (Continued)

Mnem	Name	Function
FSNE	Skip On Non-Zero	Skips the next sequential word if the Z flag of the FPSR is 0.
FSNER	Skip On No Error	Skips the next sequential word if bits 1-4 of the FPSR are all 0.
FSNM	Skip On No Mantissa Overflow	Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is 0.
FSNO	Skip On No Overflow	Skips the next sequential word if the overflow (OVF) flag of the FPSR is 0.
FSNOD	Skip On No Overflow And No Zero Divide	Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNU	Skip On No Underflow	Skips the next sequential word if the underflow (UNF) flag of the FPSR is 0.
FSNUD	Skip On No Underflow And No Zero Divide	Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNUO	Skip On No Underflow And No Overflow	Skips the next sequential word if both the underflow (UNF) flag and the overflow (OVF) flag of the FPSR are 0.
FSS, FSD	Subtract (FPAC from FPAC)	Subtracts the floating point number in one FPAC from the floating point number in another FPAC.
FSST	Store Floating Point Status	Copies the contents of the FPSR to two memory locations.
FSTS, FSTD	Store Floating Point	Copies the contents of a specified FPAC into memory.
FTD	Trap Disable	Sets the trap enable flag of the FPSR to 0.
FTE	Trap Enable	Sets the trap enable flag of the FPSR to 1.

LOGICAL MANIPULATION

Logical Format

We represent logical entities as individual bits in a 16-bit word. Each bit is treated as a separate binary value. When two words are involved (logical AND or XOR, for example) only corresponding bits of each word interact. Examples of logical operations include:

- forming the logical AND of two words;
- forming the logical complement of a word;
- shifting the contents of a word left or right.

Logical Operation Instructions

All of the logical operations instructions are shown in the following table. The *Load Effective Address* and *Extended Load Effective Address* instructions are the short and long form, respectively, of the same instruction. The short form is 16 bits in length and can directly specify a memory address from 0 to 255 or can directly specify a small area in memory surrounding the present value of the program counter or an accumulator. Long form instructions are 32 bits in length; they can directly specify any address from 0 to 77777_8.

LOGICAL OPERATION INSTRUCTIONS

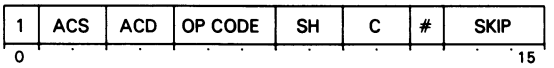
Mnem	Name	Function
ANC	AND With Complemented Source	Forms the logical AND of the contents of one accumulator and the logical complement of the contents of another accumulator.
AND	AND	Forms the logical AND of the contents of two accumulators.
ANDI	AND Immediate	Forms the logical AND of a 16-bit number contained in the instruction and the contents of an accumulator.
COM	Complement	Forms the logical complement of the contents of an accumulator.
DHXL	Double Hex Shift Left	Shifts the 32-bit contents of two accumulators left 1 to 4 hex digits depending on the value of a 4-bit number contained in the instruction.
DHXR	Double Hex Shift Right	Shifts the 32-bit contents of two accumulators right 1 to 4 hex digits depending on the value of a 4-bit number contained in the instruction.
DLSH	Double Logical Shift	Shifts the 32-bit contents of two accumulators left or right depending on the contents of a third accumulator.
HXL	Hex Shift Left	Shifts the contents of an accumulator left 1 to 4 hex digits depending on the value of a 4-bit number contained in the instruction.
HXR	Hex Shift Right	Shifts the contents of an accumulator right 1 to 4 hex digits depending on the value of a 4-bit number contained in the instruction.
IOR	Inclusive OR	Forms the logical inclusive OR of the contents of two accumulators.
IORI	Inclusive OR Immediate	Forms the logical inclusive OR of a 16-bit number contained in the instruction and the contents of an accumulator.
LEF, ELEF	Load Effective Address	Places an effective address in an accumulator.
LSH	Logical Shift	Shifts the contents of an accumulator left or right depending on the contents of another accumulator.
XOR	Exclusive OR	Forms the logical exclusive OR of the contents of two accumulators.
XORI	Exclusive OR Immediate	Forms the logical exclusive OR of a 16-bit number contained in the instruction and the contents of an accumulator.

ALC MANIPULATION

ALC Format

Each of the eight Arithmetic/Logic Class (ALC) instructions performs a specific function upon the contents of one or two accumulators and carry. The eight functions are *Add, Subtract, Negate, Add Complement, Move, AND, Complement,* and *Increment.* The instructions are identified by the mnemonics of the eight functions, which are **ADD, SUB, NEG, ADC, MOV, AND, COM,** ("**INC**").

In addition to the specific functions performed by an individual instruction, there is a group of general functions all ALC instructions can perform. These general functions include shift operations, which rotate the data left or right, or swap the bytes. Also included are various tests that can be performed on the data. With each test the instructions can check the data for some condition and skip or not skip the next sequential word, depending on the outcome of the test. Finally, the instructions can load or not load the results of the specific and general functions into the destination accumulator and carry. The diagram below shows the format of the ALC instructions.



ALC Instructions

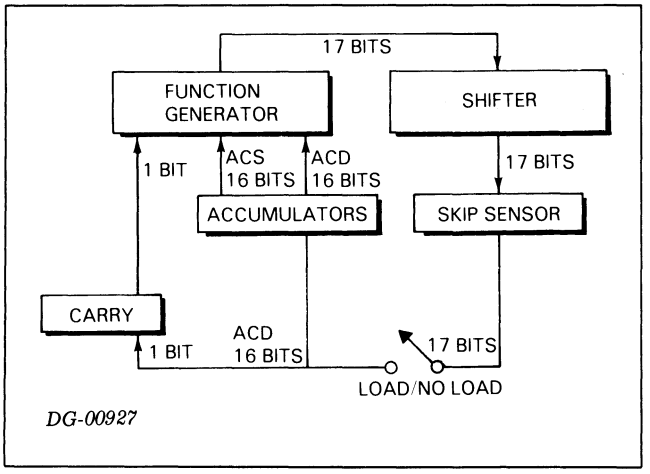
The ALC instructions are listed below.

ALC Instructions

Mnem	Name	Function
ADC	Add Complement	Adds an unsigned integer to the logical complement of another unsigned number.
ADD	Add	Adds contents of one accumulator to the contents of another.
AND	AND	Forms the logical AND of the contents of two accumulators.
COM	Complement	Forms the logical complement of the contents of an accumulator.
INC	Increment	Increments the contents of an accumulator.
MOV	Move	Moves the contents of an accumulator through the ALU.
NEG	Negate	Forms the two's complement of the contents of an accumulator.
SUB	Subtract	Subtracts contents of one accumulator from the contents of another.

ALC Instruction Execution

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



When an ALC instruction begins execution, it loads the contents of carry 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 carry. This new value is based upon three things: the old value of 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 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 unchanged
[c]=Z	01	Initialize carry to 0
[c]=O	10	Initialize carry to 1
[c]=C	11	Complement carry

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 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 $2^{16} - 1$. An overflow results. In this situation, the ALU saves the low-order 16 bits of the function result, but it complements the value of carry calculated above.

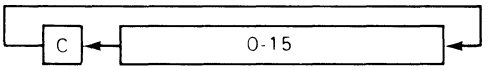
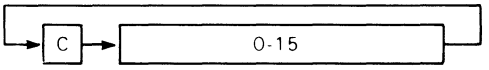
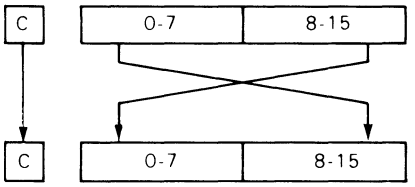
NOTE: *At this stage of operation, the ALU does not load either the saved value of the function result into the destination accumulator, or the calculated value of carry into carry.*

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 carry).

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 and ALC operation result
[sh]=R	10	Rotate right the 17-bit combination of carry and ALC operation result
[sh]=S	11	Swap the two 8-bit halves of the ALC operation result without affecting carry

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 is not affected 

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 is zero
[skip]=SNC	011	Skip if carry 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 is zero
[skip]=SBN	111	Skip if both ALC result and carry 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 carry into carry. 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 carry into carry, 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 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. This bit combination is used to specify other non-ALC instructions.*

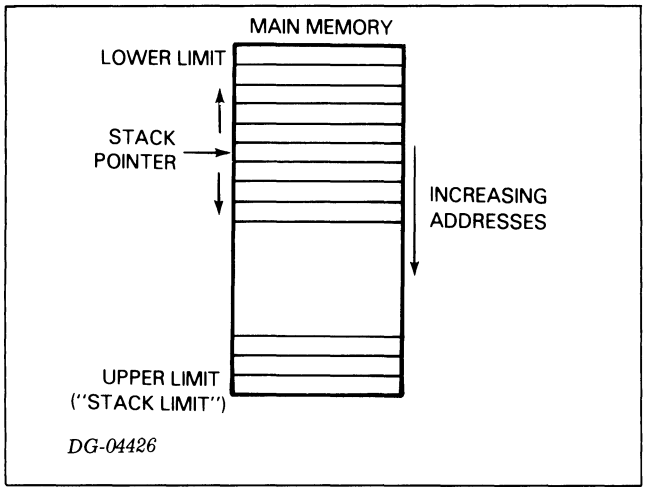
THE STACK

The stack is a series of consecutive locations in memory. In their simplest form, stack instructions add items in sequential order to the top of the stack and retrieve them in the reverse order. Several stack areas may be defined by the program, but only one stack may be in use at any time. The ECLIPSE S/250 uses the push-down stack concept to provide easily accessible temporary storage of data, variables, return addresses, etc.

The simplest use of the stack is for temporary storage of the contents of up to four accumulators, which can be stored or retrieved with one instruction. More commonly, the stack is used to store a *return block* which greatly simplifies the process of entering and returning from subroutines.

The return block can take several forms, but it usually consists of five words: the contents of the four accumulators in the first four words, and the program counter and carry in the last word pushed.

Three parameters define a stack: (1) the lower limit, or starting location; (2) the upper limit, or stack limit; and (3) the present top of the stack, or stack pointer. The lower and upper limits define the area in memory which is reserved for the stack, and the stack pointer defines the location of the last word placed onto the stack (or the next word available from the stack). A diagram of a stack area is shown below:



To use the stack, define the upper and lower limits, then use the stack instructions to put items on *(push onto)* or remove items from *(pop off)* the top of the stack. It is not necessary to keep track of the location of the top of the stack. This is done automatically by one of the stack control words (stack pointer).

Stack Control Words

The locations and uses of the stack control words are discussed in detail below:

Stack Pointer

The stack pointer contains the address of the current top of the stack. As you do push or pop operations, the value of the stack pointer changes so that it always points to the top word of the stack. A push operation increments the stack pointer contents by one, then stores the word you want to push in the new location specified by the stack pointer. A pop operation takes the contents of the word addressed by the stack pointer and loads them into a register, then decrements the stack pointer value by 1.

When you set up the stack, you usually set the value of the stack pointer to be one less than the address of the first stack word.

Location 40_8 contains the current value of the stack pointer.

Frame Pointer

Unlike the stack pointer, the frame pointer does not change its value when push and pop operations occur. If you set the frame pointer to contain the original value of the stack pointer, you have a useful reference to the first stack location.

The *Save* and *Return* instructions use the frame pointer to save the value of the stack pointer when entering or exiting subroutines. Since the frame pointer remains unchanged, it allows you to call a subroutine, perform some operation, then return to the calling program without destroying the value of the stack pointer. This means you can restore the original state of the calling program when you return from the subroutine call.

Location 41_8 contains the value of the frame pointer.

Stack Limit

The stack limit contains the upper limit of the stack area. Each push operation compares the stack pointer with the stack limit to check if there is space enough to allow the push. If the stack pointer is greater than the stack limit, then you have exceeded the size of the stack (overflow condition). For more information, see the next section on Stack Protection.

Location 42_8 contains the value of the stack limit.

Stack Fault Address

If you cause an overflow or underflow, control transfers to the Stack Fault routine. For more information, see the next section on Stack Protection.

Location 43_8 contains the (possibly indirect) address of the stack fault routine.

Stack Protection

You can enable protection for two stack error conditions: *overflow* and *underflow.*

Stack Overflow

Stack overflow occurs when a program pushes data into the area beyond that allocated for the stack, i.e., beyond the stack limit. If this occurs, data may be pushed into areas which are reserved for other purposes, possibly overwriting other data or instructions.

Overflow protection is provided by the stack limit. If a stack instruction pushes data onto the stack beyond the stack limit, a return block is pushed onto the stack, and control is transferred to the stack fault handler. To disable overflow protection, the stack limit should be set to 177777_8.

To be meaningful, the stack limit must be 10 to 23 addresses lower than the last word you reserve for the stack, because stack overflow is detected only at the end of a push operation (except in the case of the *Save* and the *Modify Stack Pointer* instructions - see details in Chapter V). Thus, it is possible to push a 5- to 18-word return block starting at the stack limit. Stack overflow will not be sensed until the last word of the return block is pushed. After the last word is pushed, stack overflow will be detected, and another 5-word return block will be pushed by the stack overflow mechanism before control is transferred to the stack fault routine. Depending on the size of the initial return block (from the normal 5 words up to the 18 words used by the floating point instruction set), the potential overflow can be 10 to 23 words long.

Stack Underflow

Stack underflow occurs when a program pops data from the area below that allocated for the stack (i.e., pops more words off than were pushed on). If this occurs, the program will be operating with incorrect and unpredictable information. Furthermore, it is possible that the program will push data into the underflow area, overwriting data or instructions.

For underflow protection to be enabled, the area allocated to the stack must begin at location 401_8 and the stack pointer must be initialized to 400_8. If the stack pointer is less than 400_8 after a pop operation, an underflow condition exists and a stack fault occurs.

Underflow protection can be disabled in two ways:

- Start the stack at a location greater than 401_8. A stack fault will not occur then unless the program underflows the stack and then continues to pop words off the stack until the stack pointer is less than 400_8. Note that this does not completely disable underflow protection - it is always possible to pop enough words off the stack to underflow it.
- Set bit 0 of both the stack pointer and the stack limit to 1. If this is done, all or a portion of the stack may reside in page zero (locations $0\text{-}377_8$), or the stack may underflow into page zero, without interference from the stack underflow mechanism.

Stack Protection Faults

Stack Overflow Protection

The *Save* and the *Modify Stack Pointer* instructions check for overflow before executing. For every other instruction that pushes data onto the stack, a check is made for overflow after the execution of the instruction. In both cases, the stack pointer and stack limit are treated as unsigned 16-bit integers and compared. If overflow has occurred, the processor:

- sets bit 0 of the stack pointer to 0;
- sets bit 0 of the stack limit to 1;
- pushes a return block onto the stack;
- executes a *jump indirect* to the stack fault address.

Bit 0 of the stack pointer and stack limit are set as indicated so that the stack limit will (temporarily) be larger than the stack pointer. In this way, the return block pushed by the overflow mechanism itself will not be interpreted as yet another overflow fault, causing a loop condition. The program counter in the return block points to the instruction immediately following the stack instruction that caused the fault.

Stack Underflow Protection

After every operation that pops data off the stack, a check is made for underflow. If the stack pointer is less than 400_8 , and bit 0 of the stack limit is 0, a stack underflow condition exists. In that case, the processor:

- sets the stack pointer equal to the stack limit;
- sets bit 0 of the stack pointer to 0;
- sets bit 0 of the stack limit to 1;
- pushes a return block onto the stack;
- executes a *jump indirect* to the stack fault address.

Bit 0 of the stack pointer and stack limit are set as indicated so that the stack limit will (temporarily) be larger than the stack pointer. In this way, the return block being pushed onto the stack by the underflow mechanism (starting at the stack limit) will not cause an overflow fault. The program counter in the return block points to the instruction immediately following the stack instruction that caused the fault.

Stack Fault Handler

The stack fault handler (created by the programmer) determines the nature of the fault. It also resets the appropriate values, and takes any other appropriate action, such as allocating more stack space or terminating the program. Note that the stack fault handler must reset bit 0 of the stack pointer and stack limit to their original values.

Initializing the Stack Control Words

Initialize the stack control words before the first operation on the stack is performed. The rules for this are as follows:

Stack Pointer

- Initialize the stack pointer to the beginning address of the stack minus one.
- If stack underflow protection is desired, initialize the stack pointer to 400_8 and start the stack area at 401_8.
- If stack underflow protection is not desired, start the stack at some location greater than 401_8.
- If you want to have all or a portion of the stack area in page zero, or you want to disable underflow protection, set bit 0 of both the stack pointer and the stack limit to 1.

Stack Limit

- Initialize the stack limit to a value greater than the stack pointer.
- If stack overflow protection is desired, initialize the stack limit to the last address allocated for the stack minus at least 10_{10}.
- If stack overflow protection is not desired, initialize the stack limit to 77777_8.
- If you want to have all or a portion of the stack area in page zero, set bit 0 of both the stack pointer and the stack limit to 1.

Stack Fault Address

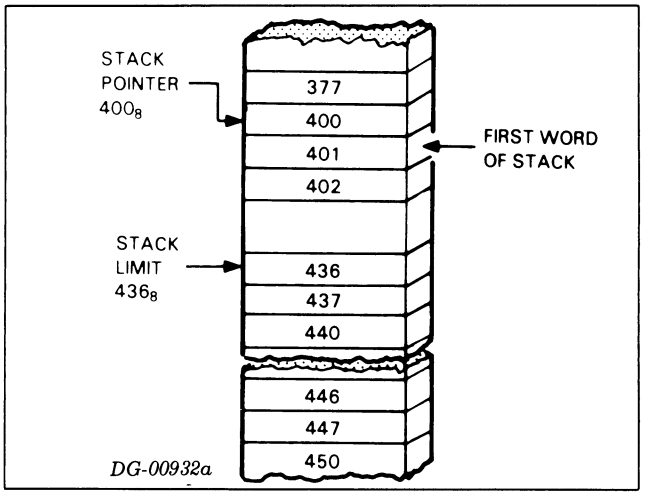
Initialize the stack fault address to the address of the routine that is to receive control in the event of a stack overflow or underflow. Bit 0 may be set to 1 to indicate an indirect address.

Frame Pointer

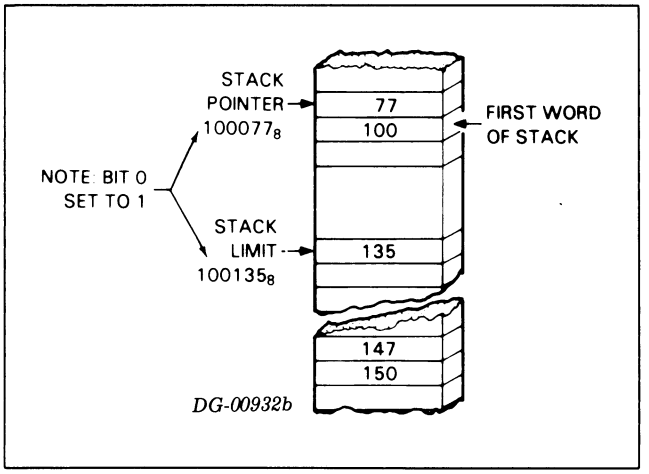
It is meaningless to attempt initialization of the frame pointer until it is actually used. The frame pointer will have no meaning until the first use of the *Save* instruction.

Examples

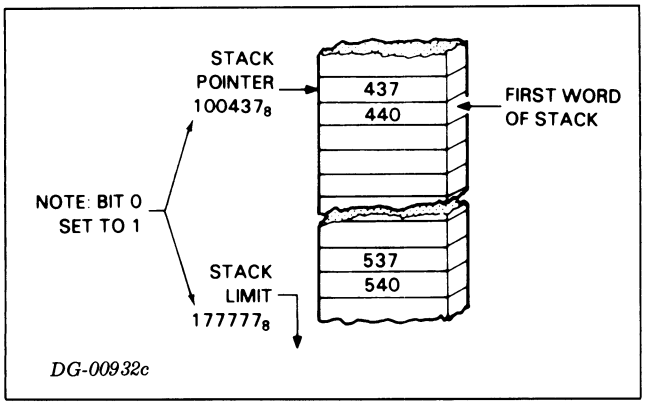
Stack area 50_8 words with underflow protection:



Stack area 50_8 words in page zero with overflow protection:



Stack area 100_8 words, no protection:



You can set up a typical stack using the following assembly language instructions:

```
        .NREL

        .EXTN   FAULT       ;Overflow handler
STACK:  .BLK    50          ;Define a 50-word stack
LIMIT:  .BLK    10.         ;Define buffer zone


SETUP:  ELEF    0,STACK-1
        STA     0,40        ;Set up stack pointer
        ELEF    0,STACK
        STA     0,41        ;Set up frame pointer
        ELEF    0,LIMIT
        STA     0,42        ;Set up stack limit
        ELEF    0,FAULT
        STA     0,43        ;Set up stack fault handler
                            ; address
        .END
```

Stack Instructions

The instructions that affect the stack are listed below.

STACK INSTRUCTIONS

Mnem	Name	Function
FPOP	Pop Floating Point State	Pops an 18-word floating point return block off the stack.
FPSH	Push Floating Point State	Pushes an 18-word floating point return block onto the stack.
MSP	Modify Stack Pointer	Changes the value of the stack pointer and checks for overflow.
POP	Pop Multiple Accumulators	Pops 1 to 4 words off the stack and places them in the indicated accumulators.
POPB	Pop Block	Returns control from a *System Call* routine or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction.
POPJ	Pop PC And Jump	Pops the top word off the stack and places it in the program counter.
PSH	Push Multiple Accumulators	Pushes the contents of 1 to 4 accumulators on the stack.
PSHJ	Push Jump	Pushes the address of the next sequential instruction on the stack and places an effective address into the program counter.
PSHR	Push Return Address	Pushes the address of the PC, plus 2, onto the stack.
RSTR	Restore	Returns control from certain types of I/O interrupts.
RTN	Return	Returns control from subroutines that issue a *Save* instruction at their entry points.
SAVE	Save	Saves the information required by the *Return* instruction.
SYC	System Call	Pushes a return block and indirectly places the address of the *System Call* handler in the program counter.
VCT	Vector on Interrupting Device Code	Performs various interrupt functions. See the I/O section in this chapter.

RESERVED STORAGE LOCATIONS

The following are reserved storage locations in the ECLIPSE S/250. These locations are used for specific functions by the CPU and should not be used for other functions.

The addresses of these locations, their names, and their functions are given below. The notation *indirectable* means that bit 0 may be set to indicate that this is an indirect address.

The following locations are in unmapped logical address space:

Loc	Name	Function
0	I/O RETURN ADDRESS	Return address from I/O interrupt; first instruction of Auto-restart routine
1	I/O HANDLER ADDRESS	Address of the I/O interrupt handler (indirectable)
2	SC HANDLER ADDRESS	Address of the *System Call* instruction handler (indirectable)
3	PF HANDLER ADDRESS	Address of the protection fault handler (indirectable)

The following locations may be in unmapped logical address space or in Map A or Map B logical address space. They are used by the VCT instruction.

Loc	Name	Function
4	VECTOR STACK POINTER	Address of the start of the vector stack (not indirectable)
5	CURRENT MASK	Current interrupt priority mask
6	VECTOR STACK LIMIT	Address of the last normally usable location in the vector stack
7	VECTOR STACK FAULT ADDRESS	Address of the vector stack fault handler (indirectable)

The following locations are in the same address space as the instructions using them:

Loc	Name	Function
20-27	AUTO-INC0 through AUTO-INC7	Auto-incrementing locations
30-37	AUTO-DEC0 through AUTO-DEC7	Auto-decrementing locations
40	STACK POINTER	Address of the top of the stack (not indirectable)
41	FRAME POINTER	Address of the frame reference within the stack (not indirectable)
42	STACK LIMIT	Address of the last normally usable location in the stack (not indirectable)
43	STACK FAULT ADDRESS	Address of the stack fault handler (indirectable)
44	XOP ORIGIN ADDRESS	Address of the start of XOP (not indirectable)
45	FLOATING POINT FAULT ADDRESS	Address of the floating point fault handler (indirectable)
46-47	RESERVED LOCATIONS	

PROGRAM EXECUTION

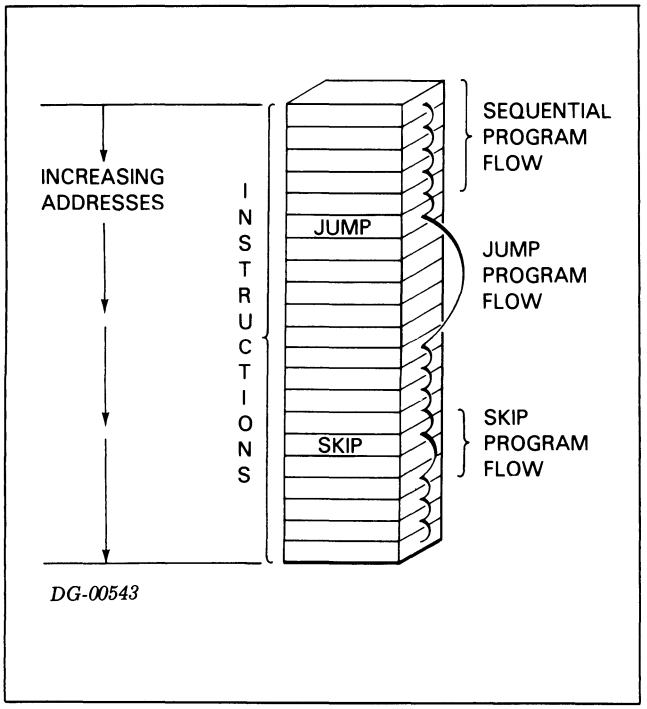
Sequential Operation

A 15-bit register called the *program counter* always contains the address of the instruction currently being executed. The program counter is incremented by one after each instruction. It can normally address the complete logical address space, i.e., 0 through 77777_8, inclusive, a total of 32,768 word locations. The address after 77777_8 is 0, and no indication is given when the counter rolls from 77777_8 to 0 in the course of sequential processing.

Program Flow Alteration

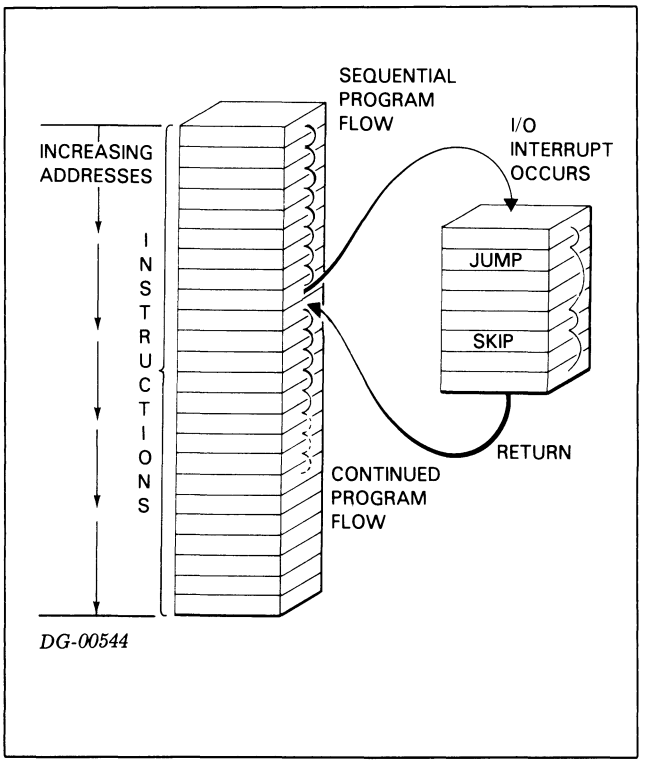
You can alter the program flow from sequential operation in two ways. Jump instructions alter the program flow by inserting a new value into the program counter. Conditional skip instructions alter the program flow by incrementing the program counter an extra time if a specified test condition is true. In either case, sequential operation continues with the instruction addressed by the updated value of the program counter.

NOTE: *Do not use a conditional skip immediately before a 2-word instruction. The conditional instruction causes a 1-word skip, which results in an attempt to execute the second word of the instruction as a 1-word instruction.*



Program Flow Interruption

The normal flow of a program may be interrupted by external or exceptional internal conditions, such as I/O interrupts or MAP faults. When this occurs, the contents of the program counter is saved, so that after the interrupt is serviced, control will return to the right place. The address of the starting instruction for the proper fault or interrupt handler is then placed in the program counter and sequential operation continues within that program. When the fault or interrupt handler has serviced the interrupt, control is returned to the interrupted program at the saved address.



Program Flow Alteration Instructions

Program flow alteration and conditional instructions are shown in the following tables.

In the first table, several instructions have both short and long forms. The short form is 16 bits in length and can directly specify a memory address from 0 to 255 or can directly specify a small area in memory surrounding the present value of the program counter or an accumulator. Long form instructions are 32 bits in length; they can directly specify any address from 0 to 77777_8.

The second table summarizes the skip instructions that test condition codes in the floating point status register.

The third table summarizes the condition tests available for the **SKP** *[t]* instruction. (This instruction tests condition codes of a peripheral device, the power-fail monitor or the interrupt system.)

The fourth table summarizes *skip* options of the ALC instructions.

PROGRAM FLOW ALTERATION INSTRUCTIONS

Mnem	Name	Function
CLM	Compare To Limits	Compares a signed integer with two other numbers and skips if first integer is between the other two.
DSPA	Dispatch	Compares a signed integer with two other numbers and continues sequential execution if the integer is not between the others; otherwise, uses the integer as an index into a table and places indexed value in the program counter.
DSZ, EDSZ	Decrement And Skip If Zero	Decrements the addressed word, then skips if the decremented value is zero.
ISZ, EISZ	Increment And Skip If Zero	Increments the addressed word, then skips if the incremented value is zero.
JMP, EJMP	Jump	Places an effective address in the program counter.
JSR, EJSR	Jump To Subroutine	Increments program counter and stores incremented value in AC3; then places a new address in the program counter.
POPB	Pop Block	Pops a return block off of the stack.
POPJ	Pop PC And Jump	Pops the top word off the stack and places it in the program counter.
PSHJ	Push	Pushes the address of the next sequential instruction onto the stack and places a new address in the program counter.
RSTR	Restore	Returns control from I/O interrupt handlers that use the stack change facility of the **VCT** instruction.
RTN	Return	Returns control from a subroutine entered via *Save* instruction.
SGE	Skip If ACS Greater Than Or Equal To ACD	Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

Program Flow Alteration Instructions (Cont'd)

Mnem	Name	Function
SGT	Skip If ACS Greater Than ACD	Compares two signed integers in accumulators; skips if first is greater than the second.
SKP *[t]*	I/O Skip	Skips if the I/O condition *t* is true.
SNB	Skip On Nonzero Bit	References a single bit in memory via bit pointer; skips if bit is 1.
SYC, SCL, SVC	System Call	Turns the MAP off if on. Pushes a return block onto the stack places address of *System Call* handler in program counter.
SZB	Skip On Zero Bit	References a single bit in memory via bit pointer; skips if bit is 0.
SZBO	Skip On Zero Bit, Set To 1	References a single bit in memory via bit pointer; skips if bit is 0 and also sets the bit to 1.
VCT	Vector On Interrupting Device Code	Identifies highest priority interrupt; passes control through a table to a handler routine for device.
XOP XOP1	Extended Operation	Pushes a return block onto the stack, indexes into the XOP table and transfers control to another procedure.
XCT	Execute	Executes contents of an accumulator as an instruction.

FLOATING POINT SKIP TESTS

Mnem	Name	Function
FNS	No Skip	The next sequential word is executed.
FSA	Skip Always	The next sequential instruction is skipped.
FSEQ	Skip On Zero	Skips the next sequential word if the Z flag in the FPSR is 1.
FSGE	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if the N flag of the FPSR is 0.
FSGT	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if both the Z and N flags of the FPSR are 0.
FSLE	Skip On Less Than Or Equal To Zero	Skips the next sequential word if either the Z flag or the N flag of the FPSR is 1.
FSLT	Skip On Less Than Zero	Skips the next sequential word if the N flag of the FPSR IS 1.
FSND	Skip On No Zero Divide	Skips the next sequential word if the divide by zero (DVZ) flag of the FPSR is 0.
FSNE	Skip On Non-Zero	Skips the next sequential word if the Z flag of the FPSR is 0.
FSNER	Skip On No Error	Skips the next sequential word if bits 1-4 of the FPSR are all 0.
FSNM	Skip On No Mantissa Overflow	Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is 0.
FSNO	Skip On No Overflow	Skips the next sequential word if the overflow (OVF) flag of the FPSR is 0.
FSNOD	Skip On No Overflow And No Zero Divide	Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNU	Skip On No Underflow	Skips the next sequential word if the underflow (UNF) flag of the FPSR is 0.
FSNUD	Skip On No Underflow And No Zero Divide	Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNUO	Skip On No Underflow And No Overflow	Skips the next sequential word if both the underflow (UNF) flag and the overflow (OVF) flag of the FPSR are 0.

SYMBOL	VALUE	TEST
[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

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

EXTENDED OPERATION FEATURE

The extended operation feature (XOP) provides an efficient method of transferring control to and from procedures. It enables the user to transfer control to any one of 32 procedure entry points (if WCS is enabled) or 48 entry points (if WCS is not enabled).

Extended Operation Instuctions

There are two extended operation instructions in the ECLIPSE S/250 instruction set.

EXTENDED OPERATION INSTRUCTIONS

Mnem	Name	Function
XOP	Extended Operation	Pushes a return block on the stack, placing the address in the stack of the specified accumulators into AC2 and AC3, and transfers control to one of 32 other procedures via the XOP table.
XOP1	Extended Operation	Same as XOP except that 32 is added to the entry number before entering the XOP table, and only 16 table entries can be specified.

MEMORY ALLOCATION AND PROTECTION

MAP Functions

NOTE: *In the following section, "MAP" refers to the Memory Allocation and Protection unit, whereas "map" refers to a set of memory translation functions used by the MAP.*

The ECLIPSE S/250 MAP unit provides the hardware necessary to control and use more than 64 Kbytes of physical memory. In addition, the MAP provides protection functions which help protect the integrity of a large system.

A MAP unit gives several users access to the resources of the computer by dividing the memory space available into blocks assigned to each user. Each time a user accesses memory, the MAP translates the address the user sees *(a logical address)* to an address the memory sees *(a physical address)*. This is all transparent to the user, and with software to control the priorities of the MAP and the CPU, several users can use the computer without being aware of the presence of the others.

For the purposes of this discussion, we define certain words and phrases:

Logical Address - The address used by the user in all programming. The logical address space is 32,768 words long and is addressed by a 15-bit address.

Physical Address - The address used by the MAP to address the physical memory. The maximum size of the physical address space is 2,097,152 words (2M) and it is addressed by a 20-bit address.

Address Translation - The process of translating logical addresses into physical addresses.

Memory Space - The addresses (physical or logical) assigned to a particular user.

Page - 1024 (2000_8) words in memory.

User Map - The set of memory address translation functions defined for a particular user.

Data Channel Map - The set of address translation functions defined for the memory references of a data channel used by a particular device.

Supervisor - The section of the operating system (software) which controls system functions such as the operation of the MAP.

Address Translation

The primary function of the MAP is address translation. The map divides each user's logical address space into 1024-word pages and correlates each logical

page with a corresponding physical page. The address space the user sees is unchanged, but the map now translates each logical address into a physical address before memory is actually accessed.

Note that there is no requirement that the physical pages assigned to a user be in any particular order in physical memory. The supervisor can therefore use physical memory very flexibly, selecting unused pages for a new user without concern for maintaining any particular arrangement. Very complete use of the physical memory is also possible, since no contiguous blocks of memory larger than 1024 words are required.

Sharing of Physical Memory

The MAP in the ECLIPSE S/250 is also capable of declaring a section of physical memory accessible to several users at once. This is useful if several users need a routine to perform some common function (e.g., trigonometric tables). Without this capability, each user would require a separate copy of the routine, thus creating many duplicate copies and wasting considerable space.

Types of Maps

Two types of maps are provided in the ECLIPSE S/250. *User maps* translate logical addresses to physical addresses when memory reference instructions are encountered in the user's program. *Data channel maps* translate logical addresses to physical addresses when data channel devices address the memory.

Each user requires a separate user map. The MAP can hold two user maps, but only one can be enabled at any one time. Thus if there are two users, the user map for each is specified and loaded into the MAP. The supervisor can then enable one or the other as needed. If there are more than two users, new user maps must be loaded as needed. In some operating systems, the operating system itself uses one of the user maps, so that a new user map must be loaded each time another user is serviced. This is not as much of an overhead burden as it sounds, because the *Load Map* instruction loads a complete map with one instruction, using relatively little time.

Separate data channel maps are needed because data channel devices can access memory without direct control from the user's program. There is thus no assurance that the proper user map would still be enabled at the time of the data channel request. The MAP can hold four data channel maps. Enabling data channel mapping enables all four data channel maps at the same time. The choice of which map is used for data channel references is made by the I/O controller making the reference. Those controllers not equipped to make this distinction use data channel map A by default. See the *Programmer's Reference Manual - Peripherals (DGC No. 015-000021).*

Unmapped Mode

So far we have assumed operation in the mapped mode. The MAP can also operate in the unmapped mode. This mode is used for diagnostic purposes and for certain MAP control functions. In unmapped mode, addresses in the range 0-75777_8 (which form logical pages 0-30) are not translated. In unmapped mode, addresses in the range 76000-77777_8 are translated by the special map for logical page 31. This allows you to access selected portions of user space while in unmapped mode.

MAP Protection Capabilities

In addition to its address translation functions, the MAP also provides protection functions. These generally protect the integrity of the system by preventing unauthorized access to certain parts of memory or to I/O devices. For example, if a set of trigonometric functions is stored in a section of memory accessible to all users, this section can be *write protected* so that users can read the functions but cannot change them.

The various types of protection available in the ECLIPSE S/250 are discussed separately below.

Validity Protection

Validity protection protects a user's memory space from inadvertent access by another user, thereby preserving the integrity and privacy of the user's memory space. When a user's map is specified, the blocks of logical addresses required by the user's program are linked to blocks of physical addresses. The remaining (unused) logical blocks are declared invalid to that user, and an attempt to access them will cause a validity protection fault.

Validity protection is always enabled, so the supervisor's responsibility is limited to declaring the appropriate blocks of logical addresses invalid.

Write Protection

Write protection permits users to read the protected memory addresses, but not to write into them. In this way, the integrity of common areas of memory can be protected. An attempt to write into a write protected area of memory will cause a protection fault.

A block of addresses is write protected when the map is specified. Write protection can be enabled or disabled by the supervisor.

Indirect Protection

An indirection loop occurs when the effective address calculation follows a chain of indirect addresses and never finds a word with bit 0 set to 0. Without indirect protection, the CPU would be unable to proceed with any further instructions, thus effectively halting the system.

With indirect protection enabled, a chain of 15 indirect references will cause a protection fault. Indirect protection can be enabled or disabled by the supervisor.

I/O Protection

I/O protection protects the I/O devices in the system from unauthorized access. In many systems, all I/O operations are performed through operating system calls. Clearly, it is undesirable to permit individual users to execute I/O instructions, since this will interfere with the operating system. If a user with I/O protection enabled attempts to execute an I/O instruction, a protection fault will occur. I/O protection can be enabled or disabled by the supervisor.

MAP Protection Faults

When a user attempts to violate one of the enabled types of protection, a protection fault occurs, as follows:

- The current user map is disabled.
- A 5-word return block is pushed onto the system stack.
- Control is transferred to the protection fault handler, through an indirect jump via location 3.

The system programmer must supply the protection fault handler. It determines the type of fault that occurred (using the *Read Map Status* instruction), and then takes the appropriate action.

A protection fault can occur at any point during the execution of an instruction. Therefore, the return address in the fifth word of the return block is not always correct. For I/O protection faults, however, the fifth word will always be the logical address of the instruction following the instruction that caused the fault.

Load Effective Address Mode

The *Load Effective Address* (LEF) instruction has the same format as some of the I/O instructions. The MAP therefore has a *Lef* mode bit which determines whether an I/O format instruction will be interpreted as an I/O or a LEF instruction. When the *Lef* mode bit is 1 (*Lef* mode enabled), all I/O format instructions are interpreted as *Load Effective Address* instructions. When the *Lef* mode bit is 0, all I/O format instructions are interpreted as I/O instructions.

The *Load Effective Address* instruction is very useful for quickly loading a constant into an accumulator. In addition, a user operating in the *Lef* mode is effectively denied access to any I/O devices, because all I/O and *Lef* instructions are interpreted as *Lef* instructions in this mode. Thus, *Lef* mode can be used for I/O protection. Note, however, that no indication is given if an I/O instruction is interpreted as a *Lef* instruction.

When not operating in the *Lef* mode, all *Lef* and I/O instructions are interpreted as I/O instructions. With I/O protection enabled, these instructions will cause a protection fault in the normal manner. With I/O protection disabled, the *Lef* instruction will be executed as an I/O instruction if possible.

Initial Conditions

At power up, the user maps and the data channel maps are undefined, the MAP is in unmapped mode, and unmapped logical page 31 is mapped to physical page 31.

After an *I/O Reset*, the MAP is in unmapped mode, the data channel maps are disabled, and unmapped logical page 31 is mapped to physical page 31.

MAP Instructions

The MAP instructions control the actions of the MAP. They are used by the supervisor program to change the mapping functions or check status of the various maps.

> NOTE: *MAP instructions* can *be executed in mapped mode if I/O protection and* Lef *mode are disabled for that user. When executed in mapped mode, the* Read Map Status, Initiate Page Check, *and* Page Check *instructions will return the desired information without changing the map. The* Map Single Cycle *instruction will disable the user map after the next memory reference. The remainder of the instructions will change the map while the map is enabled, with undesirable results for this user, another user, or the system as a whole.*
>
> *Enabling* Lef *mode only will convert all I/O instructions (including MAP instructions) to* Lef *instructions. The* Load Map *instruction, however, does not use the I/O format and therefore can still be executed. Enabling both* Lef *mode and I/O protection will prevent execution of the* Load Map *instruction.*

The MAP instructions are shown in the table below. All except *Load Map* are in I/O format using the device mnemonic MAP .

MAP INSTRUCTIONS

Mnem	Name	Function
DIA	Read Map Status	Reads the status of the current map.
DIC	Page Check	Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.
DOA	Load Map Status	Defines the parameters of a new map.
DOB	Map Supervisor Page 31	Specifies the physical page corresponding to logical page 31 of unmapped address space.
DOC	Initiate Page Check	Identifies a logical page; selects map without changing status.
LMP	Load Map	Loads successive words from memory into the MAP where they are used to define a user or data channel map.
NIOP	Map Single Cycle	Maps one memory reference using the last user map.

INPUT/OUTPUT

This section describes the Input/Output (I/O) of the ECLIPSE S/250. We first discuss the general operation of the system, then interrupts and the *Vector* instruction.

The ECLIPSE S/250 has a 6-bit device selection network, corresponding to bits 10-15 in the I/O instruction format. The devices are connected to this network in such a way that each device will only respond to commands sent with its own device code. With a 6-bit device code, 64 separate devices can be individually controlled. Some of these device codes are reserved for the CPU and certain processor options, but the remaining are available for referencing I/O devices. The assembler recognizes mnemonics for those devices assigned a code by Data General. A complete list of these is provided in Appendix A of this manual.

See *Programmer's Reference Manual - Peripherals (DGC No. 015-00021)* for details about programming specific devices in the I/O system.

Busy and Done Flags

I/O devices are controlled by manipulating their Busy and Done flags (but note that data channel devices require several programmed I/O instructions to be properly set up before they can be started with the flags). You can change the value of these flags using optional flag control command mnemonics appended to the instruction. 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 Done to 0. When the device has finished its operation and is ready to start another, it sets Busy to 0 and Done to 1.

Programmed I/O

Programmed I/O transfers data one word at a time under direct program control. For slow devices, such as teletypes, which transfer one character at a time and require an immediate echo, programmed I/O is the fastest method of I/O operation.

For faster devices, programmed I/O has several disadvantages. Several instructions are required for the transfer of each byte and other CPU operations must wait for the transfer to complete. Furthermore, data must be transferred to or from an accumulator, so an additional step is required if the data must be stored in or retrieved from memory.

Data Channel I/O

Data channel I/O permits data to be transferred in blocks of words, with program control necessary only at the start of the operation. The CPU stops during each word transfer but the transfer is made directly to

or from memory, so no additional steps are required. Data channel I/O is a very efficient method of transferring large blocks of data between memory and a fast I/O device. When single words or bytes are needed, however, programmed I/O is generally faster.

The maximum transfer rate for data channel I/O is as follows:

- Input: One word every 800 ns, or 1,250,000 words per second,
- Output: One word every 1400 ns, or 715,000 words per second.

At these rates, the CPU is effectively stopped. At lower rates, however, processing continues while data is being transferred.

Data channel devices are controlled in three phases. Phase I specifies the starting location in memory for the first word to be transferred. Phase II loads the two's complement of the number of words to be transferred into the machine. These two phases are done with programmed I/O instructions. Phase III consists of either a Read or a Write command, which are flag commands similar to those discussed above. Once the flag command is issued, the data transfer takes place when both the data channel device and the processor are ready. No further program control is required.

When a data channel device is ready to send or receive data, it issues a data channel request to the processor. 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 data channel between the device and memory without specific action by the program.

All requests are honored according to the relative position of the requesting devices on the I/O bus. The device requesting data channel service which is physically closest on the bus is serviced first, the next closest device next, and so on, until all requests have been honored. The synchronization of new requests occurs concurrently with the honoring of other requests. If a device continually requests the data channel, that device can prevent all devices further out on the bus from gaining access to the channel.

After handling all data channel requests, the processor then handles all outstanding I/O interrupt requests. Only then does program execution continue.

For more information on the data channel, see *Programmer's Reference Manual - Peripherals (DGC No. 014-000632)* and *User's Manual - Interface Designer's Reference (DGC No. 014-000629)*.

I/O Interrupts

The I/O interrupt system in the ECLIPSE S/250 provides a convenient method of handling programmed I/O with a minimum of overhead. Instead of polling each I/O device repeatedly to find out when it is ready to transmit or receive data, the interrupt system permits the program to ignore the I/O devices completely until one requires service. At that time, the device requests an interrupt. As soon as the processor is at an interruptible point in its processing, and has finished servicing data channel requests, it services the interrupt.

Interrupt System Definitions

Interrupt request line- - Common connection between all I/O devices and the computer. An I/O device places a request on the interrupt request line at the same time that it sets Busy to 0 and Done to 1, i.e., when it has finished a task and is ready to send or receive data. No information is placed on the line which permits the program to determine which device is requesting an interrupt. This must be done separately.

Interrupt On flag- - Flag in the CPU which 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 does not look at the interrupt request line at all, and therefore does not respond to any interrupts.

Priority mask- - Set of bits in the I/O devices that control the priority interrupt system. Each I/O device is connected to one of 16 bits in the priority mask. Some bits are connected to more than one I/O device. When a bit is set to 1, the devices connected to it cannot place a request on the interrupt request line, although they can set their Busy flags to 0 and their Done flags to 1. Since the mask can be changed by the program, different devices can be inhibited at different times to conform to the needs of a priority system.

Base level- - The state of a program when no I/O devices are inhibited (all mask bits are 0) and no interrupt processing is in progress. This is the environment in which user program execution takes place.

Nonbase level- - Any system state in which some I/O devices are inhibited and/or interrupt processing is in progress. Interrupt handlers operate at non-base level.

In the next section we will discuss interrupts. First we will discuss interrupts without a priority system, and then we will consider a priority interrupt system.

Processing an Interrupt Without a Priority System

When an I/O device completes its operation and is ready to send or receive more data, it sets its Busy flag to 0 and its Done flag to 1. Since its priority bit is 0, it also places a request on the interrupt request line. If the Interrupt On flag is 1 when the processor is next interruptable, the interrupt will be serviced.

When servicing an interrupt, the CPU first sets the Interrupt On flag to 0 so that no devices can interrupt the first part of the interrupt service routine. If a user map is enabled, it is disabled. The CPU then places the contents of the updated program counter into physical memory location 0 and jumps indirect via location 1, where it expects to find the address (direct or indirect) of the interrupt service routine.

The interrupt service routine (supplied by the user) must 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 that device as necessary.

The service routine can identify the interrupting device by using *I/O skip* instructions, or the *Interrupt acknowledge* instruction. Or it can save the return information and identify the interrupting device with one instruction by using the *Vector on interrupting device code* instruction.

The *Interrupt Acknowledge* instruction returns the 6-bit device code of the device requesting the interrupt. The *Vector* instruction, in addition to saving return information on the stack, performs an *Interrupt Acknowledge* instruction and uses the code returned as an index into a table of addresses. These addresses are the beginnings of the various device service routines.

After servicing the device, the interrupt routine should restore the saved values of the accumulators and the carry bit, set the Interrupt On flag to 1, and return to the interrupted program. The *Interrupt Enable* instruction sets the Interrupt On flag to 1, and, if the value of the flag was changed, allows the processor to execute one more instruction before the next interrupt can take place.

This next instruction should return control to the interrupted program. Since the updated value of the program counter was placed in location 0 by the CPU at the start of the interrupt service routine, a *jump indirect*, via location 0, returns control to the proper location in the interrupted program.

Priority Interrupt System

The need for a priority interrupt system can be illustrated as follows:

If the Interrupt On flag remains 0 throughout the interrupt service routine, the CPU cannot be interrupted while an I/O device is being serviced. All other devices, therefore, must wait until the first device is finished. If the Interrupt On flag is returned to 1 after the initial portion of the service routine, any I/O device can interrupt the servicing of any other I/O device. While this might be reasonable for some devices, it is not for others. It is therefore desirable to have a system of interrupt priorities which will permit some devices to interrupt certain others without disrupting the orderly processing of data.

A rudimentary sort of priority system will result from keeping the Interrupt On flag 0 throughout the service routine. The priority of the I/O devices is then determined, either by the order in which the I/O SKIP instructions poll the I/O devices, or (using the *Interrupt Acknowledge* or *Vector* instructions) by the physical location of the I/O devices on the I/O bus. Both of these methods are very inflexible, however.

The ECLIPSE S/250 has the hardware and instructions for a more flexible and efficient priority system, with up to sixteen levels of priority interrupts. The interrupt service routine has full control of this system, and can change the priorities of various devices as necessary.

Setting Up a Priority System

To set up a system of priorities, place a *Mask Out* instruction in the interrupt service routine for each device. This instruction changes the priority mask, thus controlling which devices can interrupt. All those devices which should not interrupt the device being serviced are masked out (prevented from requesting an interrupt) if their mask bits are 1. In addition, all pending interrupt requests from devices controlled by that bit are disabled. The other mask bits, corresponding to the devices which can interrupt, are set to 0.

If this is done in each interrupt service routine, then the mask will always mask out those devices which should not interrupt the device presently being serviced. This is a dynamic process, changing each time a different device is serviced, resulting in a system of priorities. The device with the highest priority will be able to interrupt all other devices, and the device with the lowest priority will be interruptible by all other devices.

Devices which operate at roughly the same speed are controlled by the same bit in the mask. Appendix A lists the mask bit assignments in addition to the device code assignments. Although the bit assignments are fixed, the priorities are set by the programmer to fit the situation and are dynamically adjustable.

A multiple priority level interrupt handler must be interruptable without damage. Usually this is not true for the initial portions of the interrupt handler, so the Interrupt On flag is initially set to 0. The interrupt handler must first save return information after receiving control. This information must 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 of the previous level.

Next, the correct service routine must be chosen. This routine must save the current priority mask and establish a new one. Once this is all completed, the *Interrupt Enable* instruction can be used to set the

Interrupt On flag to 1, enabling those devices not restricted by the priority mask to interrupt if necessary.

After servicing the interrupt, the interrupt service routine should:

- disable the interrupt system,
- reset the priority mask to the condition it was in when the routine was entered,
- restore the accumulators and the carry bit,
- enable the interrupt system,
- return control to the interrupted program.

Stack Changes

The interrupt handler usually requires use of a stack. Rather than work with the user stack, you can define a new stack which is reserved for use by the interrupt handler. This overcomes the following problems:

- There is no guarantee that a user stack will always be defined,
- The user stack pointer could be just below the stack limit. The interrupt handler would then overflow the user stack.

The stack environment should be changed whenever a transition is made from base level to non-base level or vice versa.

If an interrupt is already being processed (i.e., the program is not at base level) when another interrupt occurs, the stack environment should not be changed, since this has already been done for the first interrupt. If desired, return information to permit an easy return to processing the first interrupt can be pushed onto the new stack before the second interrupt is processed.

The *Vector* instruction handles all these stack changes by using different modes in different situations. The next section will discuss the use of this instruction.

Using the Vector Instruction

The *Vector on interrupting device code* instruction can simplify the design of an interrupt handler by doing many of the required steps in one instruction. It can also perform different levels of tasks as needed within the interrupt handler.

The *Vector* instruction has five different modes that can be used in different circumstances. The simplest of these is scarcely more complex than the *Interrupt acknowledge* instruction. It does not save any information on the state of the computer at the interrupt, and takes very little time. The most complex mode, on the other hand:

- saves considerable information on the state of the machine,
- stores the user stack parameters,
- creates a new stack,
- resets the priority mask,

and, of course, takes much longer.

When choosing which mode to use, you must weigh the importance of saving the state of the computer, having a separate vector stack, and changing the priority mask, against the time used for each interrupt. Note that you are not committed to one mode throughout the interrupt handler. It is possible to use different *Vector* instruction modes at different times to serve different needs. An example at the end of this section illustrates this.

Mode A - is used when a device requires immediate interrupt service. This would be the case for unbuffered devices with very short latency times, or for real time processes that require immediate access. The price you pay for fast reaction time is that nothing is saved to make the return from the interrupt easier.

Modes B through E - all create a priority structure which permits some interrupting devices to interrupt the service of certain others. This takes longer than mode A service, but permits devices which need immediate service to get it even if a slower device is already being serviced.

Modes D and E - both initiate a new stack. You should use them only when operating at base level (no interrupt processing in progress) since they set up a new vector stack for use by the interrupt handler and store the (old) user stack parameters in it. Once this new stack has been set up, there is no reason to try to set it up again if a new interrupt occurs before the old one was finished. Mode E also pushes a return block onto the stack to make return to the first interrupt handler easier.

Modes B and C - do not initiate a new stack, and are therefore appropriate to use when operating at non-base level (that is, when a device interrupts the interrupt processing of another device). Mode C also pushes a new return block onto the stack.

Special Mnemonics

Some of the ECLIPSE S/250 I/O instructions 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:

DOB*f* *ac*,CPU

instead of the special mnemonic:

MSKO *ac*

The special mnemonic sets bits 8 and 9 to 00.

I/O INSTRUCTIONS

Mnem	Name	Function
DIA	Data In A	Transfers data from the A buffer of an I/O device to an accumulator.
DIB	Data In B	Transfers data from the B buffer of an I/O device to an accumulator.
DIC	Data In C	Transfers data from the C buffer of an I/O device to an accumulator.
DOA	Data Out A	Transfers data from an accumulator to the A buffer of an I/O device.
DOB	Data Out B	Transfers data from an accumulator to the B buffer of an I/O device.
DOC	Data Out C	Transfers data from an accumulator to the C buffer of an I/O device.
DOC CPU	Halt	Stops the Processor.
DIB CPU	Interrupt Acknowledge	Returns the device code of an interrupting device.
INTDS (NIOC CPU)	Interrupt Disable	Sets Interrupt On flag to 0.
INTEN (NIOS CPU)	Interrupt Enable	Sets Interrupt On flag to 1.
DIC CPU	Reset	Sets all Busy and Done flags and the priority mask to 0.
DOB CPU	Mask Out	Changes the priority mask.
NIO	No I/O Transfer	Changes a flag without causing any other effect.
DIA CPU	Read Switches	Places the contents of the console data switches into an accumulator.
SKP	I/O Skip	Tests a flag and skips the next sequential word if the test condition is true.
SKP CPU	CPU Skip	Tests the Interrupt On or Power Fail flag and skips the next sequential word if the test condition is true.

Flag Control Commands

The tables below summarize the operations performed by the optional mnemonics used in I/O instructions. The first table applies to those instructions that change the Busy and done flags.

Mnem	Name	Function
[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 depends upon device.

The next table applies to the I/O Skip instruction.

Mnem	Name	Function
[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.

The last table applies to I/O instructions using the device code mnemonic CPU (device code 77_8). Instead of manipulating or testing the Busy and Done flags, these instructions operate on the Interrupt On and Power Fail flags.

Mnem	Name	Function
[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 (used only with VCT.
[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.

BASIC I/O DEVICES

There are three I/O devices which are common to all ECLIPSE S/250 Computer systems. These devices are an Asynchronous Line Controller, a Real-Time clock (RTC), and a Programmable Interval Timer (PIT).

Asynchronous Line Controller

The Asynchronous Line Controller is the interface to the primary terminal of the ECLIPSE S/250 system. It can transmit and receive serial asynchronous information at jumper selectable rates from 110 to 9600 baud. The ALC is program compatible with Data General's 4010 controller.

Real-Time Clock

The Real-Time Clock generates low frequency I/O interrupts for performing time calculations independent of CPU timing. These interrupts may be used as a time base in programs which require it. The frequency of the clock is program selectable to AC line frequency, 10Hz, 100Hz, and 1000Hz.

Programmable Interval Timer

The Programmable Interval Timer is a CPU-independent time base which can be programmed to initiate program interrupts at fixed intervals ranging from 100 microseconds to 6.5536 seconds in increments of 100 microseconds. It can also be sampled with I/O instructions at any point in its cycle to determine the time until the next interrupt. The PIT is often used in multiprogram operating systems where the timer is used to allocate CPU time to different programs on a "time slice" basis.

Programmable Interval Timer

The Programmable Interval Timer (PIT) consists of a 16-bit initial count register and a 16-bit counter. During operation, the counter is loaded with the contents of the initial count register and is then incremented at 100 microsecond intervals until the count reaches 177777_8. The PIT then initiates a program interrupt request. At the end of the next 100 microsecond interval, it is again loaded with the contents of the initial count register and the counting process is repeated. A Busy flag and a Done flag control the operation of the device.

Two instructions are used to load the initial count register, and to read the present value of the counter. The instructions are shown in the table below.

PIT INSTRUCTIONS

Mnem	Name	Function
DOA	Specify Initial Count	Selects the value which will be loaded into the counter each time the PIT is started or overflows.
DIA	Read Count	Reads the current value of the PIT counter.

Programming Considerations

In order to obtain a particular time interval between program interrupt requests, load into the initial count register the two's complement of the number of 100 microsecond intervals between interrupt requests. Whe you first start the PIT, the interval to the first program interrupt request may be anywhere from 0 to 6.5536 seconds. After the first interrupt request, the time between program interrupt requests will be the value selected by the contents of the initial count register.

Real Time Clock

The real time clock (RTC) initiates program interrupts at fixed intervals which are independent of CPU timing or programs. Four timing intervals may be selected by program control. A Busy and a Done flag control the operation of the device.

One instruction programs the real time clock, as shown in the table below.

REAL TIME CLOCK INSTRUCTION

Mnem	Name	Function
DOA	Select RTC Frequency	Selects the frequency of real time clock interrrupts.

When you first start the real time clock, the first program interrupt request 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 to 1 before the clock period expires. After power up or **IORST**, the clock is set to the line frequency. After power up, the line frequency pulses are available immediately, but five seconds must elapse before a steady pulse train is available from the clock for other frequencies.

Asynchronous Line Controller

The Asynchronous Line Controller is the communication link between the ECLIPSE S/250 computer and the system's master terminal. It supports asynchronous communication at selected rates from 110 to 9600 baud in 7-bit codes with program generated parity or 8-bit codes with no parity. One or two stop bits may be used with either format. Since the asynchronous communications input and output can generate program interrupts independently, each has its own device code and is controlled by its own set of Busy and Done flags.

A single instruction is used to program the asynchronous line input (ALI). The instruction is shown in the table below.

ALI INSTRUCTION

Mnem	Name	Function
DIA	Read Input Buffer	Reads a character from the input buffer.

A single instruction programs the Asynchronous Line Output (ALO), as shown in the table below.

ALO INSTRUCTION

Mnem	Name	Function
DOA	Load Output Buffer	Places a character in the output buffer.

The asynchronous line controller is set up to transmit and receive 8-bit characters without parity checking. You can send and receive 7-bit characters with even, odd, or mark parity under program control by using the high order bit in the 8-bit character (bit 8 in the AC) as a parity bit. On transmission, the program which drives the asynchronous line controller must calculate and insert the correct parity bit. On reception, the program must calculate and check parity on the received character.

You must also be aware of timing constraints on the receive portion of the controller. As each character is received, it is placed in an input character buffer, the Done flag is set to 1, and the Bus flag is set to 0. If the program controlling the receiver does not transfer the character before the next character is received, the contents of the input character buffer will be overwritten and the previous character will be lost. Typically, the inter-character time at 110 baud is 100 milliseconds and at 9600 baud the inter-character time is approximately 104 microseconds.

CONSOLE

The ECLIPSE S/250 console is a powerful tool for creating and debugging programs. The state of the CPU, and the floating point processor can be seen in the console's status lights at all times. The Program Load function helps load programs quickly and easily from peripheral devices. It can perform transfers using either the data channel, the optional Burst Multiplexor Channel, or programmed I/O. The rotary switches, Operation and Address Source, that appear on the right hand side of the console allow you to closely monitor your code and the peripheral devices as they access memory. With the Address Mode switch you can alter addressing mode to monitor the logical address input to the MAP, or the physical address output from the MAP.

The following section offers a brief introduction to these features and suggests some ways to use them to get their maximum benefit. See the tables in Chapter VII for complete documentation of the console facilities.

Using the Console Address Mode Feature

The ECLIPSE S/250 console has three addressing modes: logical, physical, and memory diagnostic. The Address Mode rotary switch on the right hand side of the console specifies which of these modes is active.

Logical Address Mode

In logical addressing mode the console uses and monitors only 15-bit logical addresses. All operations and functions that require an address from the console use only the 15 low order data switches. The console address lights will display the contents of the logical address bus. If the MAP is enabled, all memory addressing from the console and the program in execution will be mapped; otherwise, the address will be a physical address to the lowest 64K-byte of memory.

Physical Address Mode

In physical addressing mode the console uses and monitors 20-bit physical addresses. All console operations and functions that require an address use all 20 of the data switches. The address lights will display the contents of the physical address bus. If the MAP is enabled, memory addressing by the program in execution will be mapped. Memory addressing from the console will not be mapped. Console functions that use the 15-bit PC register (e.g. Examine Next) will prefix the PC with the contents of the 5-bit extended address (EA) register to produce a 20-bit address. (The EA register receives the contents of the 5 left-most data switches whenever a Start or Examine function is initiated.)

Memory Diagnostic Mode

Use memory diagnostic address mode only for diagnostic testing. In MD mode the MAP must be turned off or else the results of memory addressing will be undefined. The console or a program being executed can address only one contiguous 64K-byte segment of memory at any one time. The contents of the EA register define which 64K-byte segment is used. Neither the console nor an executing program can access the other segments until you change the value of the EA register or alter the Address Mode switch.

Two of the console functions alter the EA register: Examine and Start. When used, these functions place the setting of the 5 left-most data switches (X0-X4/0) into EA. Each time you initiate Examine or Start the EA is refreshed. However, that value will have no meaning until you change the Address Mode switch to MD or Phy.

Memory diagnostic mode places the contents of the EA register on the physical address bus in the 5 left-most bits. The normal logical address supplies the remaining 15 bits. Most programs should execute normally in Memory Diagnostic mode. However, this addressing mode is not recommended for normal operation.

Using the ECLIPSE S/250 Program Loader

The Program Load function performs a microdiagnostic test, then puts a 32-word bootstrap loader in locations $0\text{-}37_8$ of memory. (Appendix G contains a listing of this bootstrap loader.) Prior to initializing this function you must perform the following steps:

- Prepare the I/O device for the read (load appropriate tape or disc, turn on device, etc.);
- Set Data switches 10-15 to the device code of that device;
- If it is a data channel or burst multiplexor channel I/O device, set switch X4/0 to 1; otherwise, set it to 0;
- Set data switch 4 to enable or disable the microdiagnostic test.

The microdiagnostic program is a quick (1 second) microcode check of the low order 32 Kbyte of memory. (MD mode will cause the check to be performed in the 32 Kbyte specified by the EA register.) If data switch 4 is 1, the microdiagnostic program will not execute. If data switch 4 is 0, the microdiagnostic test will execute before the bootstrap loader is placed in memory. If the diagnostic test detects no errors, the bootstrap loader will then enter memory. (If you initiate a program load with all data switches set to 0, then the microdiagnostic will not terminate until it detects an error or you set data switch 4 to 1.)

After the bootstrap loader is in memory, it automatically begins execution at location 0. The bootstrap loader reads the data switches, creates its own I/O instructions with the specified device code, and then performs one of two program load procedures depending upon the value of Data switch X4/0.

Program Load (Data Channel, Optional Burst Multiplexor)

If switch X4/0 is a 1, the bootstrap loader starts the I/O device and loops at location 377_8 until the data transfer places a word into that location. The loader then executes it as an instruction. Typically, that word is an instruction to halt or to jump into the data that has just been transferred.

NOTE: *Some burst multiplexor and data channel devices transfer more than 256 words at a time. It is up to the device or the program to control the transfer after 256 words have been read.*

Program Load Using Programmed I/O

If data switch X4/0 is a 0, the bootstrap loader reads the program via programmed I/O. The device must supply 8-bit data bytes. The loader stores each pair of bytes as a single word in memory: the odd byte becomes the left half of the word, and the even byte becomes the right half.

The bootstrap loader ignores leading null characters. It does not begin storing any words until it reads a non-zero synchronization byte. The first word following this synchronization byte must be a two's complement number which is the negative of the total number of words to be read, including itself. The number of words to be read, including the word count, may not exceed 192_{10}.

The bootstrap loader stores these words beginning at memory location 100_8. It transfers control to the location of the last word read, after finishing the programmed read.

Debugging Programs Using the Console

The ECLIPSE S/250 console offers a number of powerful debugging features. With it you can halt program execution, examine the contents of accumulators or memory, modify code or data, and resume normal sequential execution. Additionally, you can step through a program one instruction at a time and examine or change code and data between instructions.

The Operation switch that controls Monitor, Stop on Store, and Stop on Address is a useful debugging aid. With its various settings you can monitor a particular memory location, and stop the program if it tries to read or write that location.

The Stop on Address feature may also be used like a break point by setting the data switches to the address of an instruction. When the instruction is fetched, the machine freezes, and the Match lamp lights. You can then resume normal execution with the Continue function, or you can put the machine in the halt state with the Step Instruction function. Once the halt state has been reached, you have full use of the console. (To restart the CPU, restore the PC value; then press Continue.)

A useful feature for debugging some routines is the Instruction Step function. It will execute an instruction in the same way as normal sequential execution would, but between instructions the processor is in the halt state. The console is fully operational between instructions, and the code and data may be examined or changed as needed.

Assembly language programmers seldom use the function Microinstruction Step, since the microcode is not accessible to them. However, you can micro step through a single machine instruction by following these steps:

Place the instruction in the data switches.
Push the Inst/uInst switch down.
Push and hold the PLoad/Exec switch down.
Continue to push Inst/uInst until ROM address 0002_8 appears in the ROM address lights.

The first microinstruction executed will not be part of the instruction.

To execute several machine instructions together follow these steps:

Place the instructions to be evaluated in memory.
Place the address of the first instruction in the data switches.
Push the Inst/uInst switch down.
Push the Strt/Cont switch up.
Continue to push Inst/uInst down.

The first microinstruction will not be part of the instruction at the start address. You may then step through the microcode completely, including subsequent machine instructions. In either case, normal sequential execution may be resumed at any time with the Continue function.

MEMORY ERROR CHECKING

Error Checking And Correction

The Error Checking and Correction (ERCC) facility is designed for applications requiring either a high degree of reliability for the main memory of a system, or a graceful "fail-soft" capability in the event of memory errors. The ERCC facility will detect and correct all single-bit errors that occur in memories equipped with the option. ERCC is available for semiconductor memory only.

Each ERCC memory word is 21 bits long. These 21 bits consist of 16 data bits followed by 5 ERCC check bits. Each time the CPU writes data into a location, a hardware encoder constructs the check field bits from the 16 data bits. When the CPU reads a memory location, the encoder checks the ERCC bits read from memory. If the 21 bits do not generate an error code when read, the ERCC facility passes the 16 data bits on to the CPU. If the 21 bits generate an error code, a single bit error has occurred. The memory pauses while the ERCC facility corrects the single bit in error and rewrites the entire corrected word back into the memory location. The ERCC facility then passes the data on to the CPU and requests an interrupt. If no error occurs, no time is taken and the cycle time of the memory is unchanged from its non-ERCC counterpart.

ERCC logic enables the facility to detect and correct all single-bit errors. In the rare event that a multi-bit error occurs, the facility either detects and reports it with no correction, or incorrectly interprets it as a single-bit error and complements the bit.

ERCC Instructions

The operation of the ERCC facility is governed by one I/O instruction. Two other instructions are used to interrogate ERCC after it has detected and corrected an error. ERCC contains a Done flag which is set to 1 after an error has been detected and initiates an interrupt request. A fourth instruction is used to set this flag to 0. The ERCC facility has no Busy flag and no mask bit in the priority mask. The device code for the ERCC facility is 2. The assembler recognizes the mnemonic **ERCC** for this device code.

All the ERCC instructions with the exception of the *Clear ERCC interrupt request* use an accumulator, which is specified when coding the instruction, to receive the data or contain the control information.

ERCC INSTRUCTIONS

Mnem	Name	Function
DOA	Enable ERCC	Enables the ERCC facility according to the setting of bits 14-15 of the specified accumulator.
DIA	Read Memory Fault Address	Returns the low-order bits of the memory location which has produced an error.
DIB	Read Memory Fault Code	Returns a 5-bit error code which tells which bit was in error. Also returns the high-order bits of the memory fault address.
NIOS	Clear ERCC Interrupt Request	Sets the ERCC Done flag to 0; clears an interrupt request if one was pending.

POWER FAIL/AUTO-RESTART

When power is turned off and then on again, core memory is unaltered, but the contents of semiconductor memory are lost. The state of the accumulators, the program counter, and the various flags in the CPU and SC memory then are indeterminate. The power fail 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 loss of power, sets the Power Fail flag to 1 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 (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 enough time to execute 1000 to 1500 instructions, so there 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 power switch on the front panel. If the switch is in the *on* position, the CPU remains stopped after power is restored. If the switch is in the *lock* position, then 222ms after power is restored, the CPU executes the instruction contained in physical location 0, thereby transferring control to the restart procedure.

The contents of semiconductor memory are lost under a power failure. Therefore, the auto restart facility should not attempt to restart the system, even with the power switch in the LOCK position, if the host contains semiconductor memory. This can be controlled by proper positioning of jumpers on the power fail facility. The local memory of any optional WCS, SP, IAP, and/or DCU in the system is semiconductor so the restart facility must reload those memories before restarting the processors.

Power Fail Instructions

The power fail instructions test the state of the power fail flag. They use the device code 77_8. The assembler recognizes the mnemonic CPU for this device code.

The power fail facility has no priority mask bit in the priority mask. It responds to the *Interrupt acknowledge* and *Vector* instructions with device code 0.

POWER FAIL INSTRUCTIONS

Mnem	Name	Function
SKPDN, CPU	Skip If Power Fail Flag Is One	If the Power Fail flag is 1 (i.e., power is failing), the next sequential word is skipped.
SKPDZ, CPU	Skip If Power Fail Flag Is Zero	If the Power Fail flag is 0 (i.e., power is not failing), the next sequential word is skipped.

Chapter III
OPTIONAL FACILITIES

In this chapter, we describe the optional facilities in the ECLIPSE S/250 along with the instructions that program these facilities. The optional facilities are:

- Floating Point Functions;
- Character Instructions;
- Burst Multiplexor Channel (BMC);
- Writable Control Store (WCS);
- Integral Array Processor (IAP);
- Satellite Processor (SP).

You can find complete descriptions of all the ECLIPSE S/250 instructions, other than I/O instructions, in Chapter IV. Chapter V contains complete descriptions of all the I/O instructions. Chapter VII describes all IAP instructions. For a list of SP instructions, refer to Chapter III under the heading **Satellite Processor.**

LOGARITHMIC AND TRIGONOMETRIC FUNCTIONS

Floating Point Functions

Floating point functions are used by high-level language compilers such as FORTRAN 5, DG/L or PL/I to significantly increase the speed of programs written in these languages. Each instruction performs a single numerical function, such as taking the logarithm or square root of an argument. Since the entire algorithm is implemented in microcode, the speed increase over an assembly language subroutine is significant.

Algorithm Coefficients

Many of the instructions in this section use algorithms containing one or more polynomials to perform the required function. In those cases, the instruction word must be followed by a series of polynomial coefficients for proper operation of the algorithm. The coefficients given in the tables following these instructions cause the algorithm to perform the function specified in the instruction description.

All the floating point functions are interruptible; interrupted instructions are restarted after the interrupt. As a result, certain Real Time Clock or Programmable Interval Timer frequencies may cause looping when you are evaluating very large polynomials with the *Polynomial evaluation* function. Maximum interrupt latency is 10 microseconds.

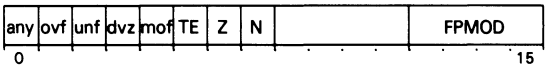
Floating Point Function Instructions

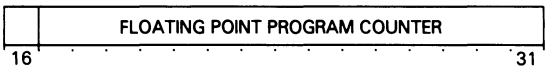
All the floating point functions are shown in the following table. Some instructions have two forms. The form using a mnemonic ending in S produces a single-precision floating point result while the form using a mnemonic ending in D produces a double-precision floating point result.

FLOATING POINT FUNCTION INSTRUCTIONS

Mnem	Name	Function
FCOSS, FCOSD	Cosine	Forms the cosine of a number.
FEXPS, FEXPD	Real Exponential	Forms the exponential of a number.
FLOGS, FLOGD	Natural Logarithm	Forms the natural logarithm of a number.
FPLYS, FPLYD	Polynomial Evaluation	Evaluates a polynomial of a specified positive degree.
FSINS, FSIND	Sine	Forms the sine of a number.
FSQRS, FSQRD	Square Root	Forms the square root of a number.

The floating point functions update the floating point status register as appropriate. Note, however, that the result of a floating point function after an exponent overflow or underflow, or after an attempt to divide by zero, is not a meaningful number. The format of the floating point status register is as follows:





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 FSCAL instruction, a significant bit was shifted out of the high order end of the mantissa; this bit is also set during a FIX instruction if the result does not fit into the destination location.
5	TE	Trap Enable - If this bit is 1, setting any of bits 1-4 will result in a floating point fault.
6	Z	Zero bit - The result of the last floating point operation was zero.
7	N	Negative bit--The result of the last floating point operation was less than zero.
8-11*	---	Reserved for future use.
12-15	FPMOD	Indicates computer series supporting the floating point instruction set. 0000 S/200, C/300, S/230, C/330 0001 S/130, S/250 standard FP 0010 M/600, C/350, S/250 optional FP 0011 Reserved for future use. 0100 Reserved for future use. 0101 8660 SP, 8661 SP 0110 C/150, S/250 standard EAU 0111- Reserved for 1111 future use.
16	---	Reserved for future use.
17-31	FPPC	Floating Point Program Counter - This is the logical address of the last floating point instruction executed. In the event of a floating point fault, this is the address of the floating point instruction that caused the fault.

**These bits are used as internal flags by the firmware; Preserve them when saving the state of the FPSR.*

CHARACTER MANIPULATION

Character Instructions

The four character instructions manipulate strings of characters. Each character in a string occupies one byte. The character instructions:

- compare one byte string to another;
- move a byte string from one area of memory to another;
- translate a character string from one data type to another.

The character instructions are described in the table below.

CHARACTER INSTRUCTIONS

Mnem	Name	Function
CMP	Character Compare	Compares one string of characters in memory to another string.
CMT	Character Move Until True	Moves a string of bytes from one area of memory to another until a table-specified delimiter character is encountered or the source string is exhausted.
CMV	Character Move	Moves a string of bytes from one area of memory to another under control of the values in the four accumulators.
CTR	Character Translate	Translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second string of bytes.

HIGH-SPEED I/O

Burst Multiplexor Channel

The Burst Multiplexor Channel (BMC) is a high speed communications pathway which transfers data directly between main memory and high speed peripherals. It is controlled by the device controller performing the data transfer. No program control or CPU interaction is required except to set up the BMC's map table. As a result, BMC data transfers are limited only by the memory speed. If the BMC and the CPU attempt to access memory at the same time, the BMC has priority.

The maximum data rate for the BMC is:

- Input: 200 nsec per word or 5 Megawords/sec.
- Output: Alternating cycle times of 200nsec per word and 400 nsec per word, or 3 1/3 Megawords/sec.

BMC Address Modes

The BMC has two address modes. In the unmapped (physical) mode, the BMC receives 20-bit addresses from the device controllers, and passes them directly to memory. As each data word is transferred to or from memory, the BMC increments the destination address, causing successive words to move to/from consecutive locations in memory.

The other BMC address mode is mapped. When a controller initiates a data transfer, it can specify the mapped (logical) mode. The high order 10 bits of the logical address form a logical page number, which the BMC MAP translates into an 10-bit physical page number. This page number, combined with the 10 low order bits from the logical address, forms a 20-bit physical address which is passed to memory.

BMC MAP

Since the BMC uses a different memory port from the CPU, it contains its own MAP. This BMC MAP uses its own map table to translate logical page numbers to physical ones. The table contains 1024 map registers, each of which holds a 10-bit physical page number (the controlling program will have loaded these physical page numbers into the table before I/O transfers begin). The BMC uses the logical page number as an index into the map table, and the contents of the selected map register become the high order 11 bits of the physical address.

Note that when the BMC performs a mapped transfer, it increments the destination address after it moves each data word. If the increment causes an overflow out of the 10 low order bits, this selects a new map register for subsequent address translation. Depending on the contents of the map table, this could mean that successive words are not transferred to/from consecutive pages in memory.

Burst Multiplexor Channel Instructions

Map loads and reads are initiated by an I/O Start command to the burst multiplexor channel. The channel's Busy flag is set to 1 when a map load or read is in progress. There is no Done flag and the burst multiplexor channel never causes program interrupts.

Device code 5 is assigned to the burst multiplexor channel. The assembler recognizes the mnemonic **BMC** for this device code.

The operation of the BMC is essentially transparent to a program executing in the host processor. The program must set up the map table, but the operation of the burst multiplexor channel and its MAP is controlled by the device controller performing the data transfer. The table below summarizes the burst multiplexor channel instructions.

BURST MULTIPLEXOR CHANNEL INSTRUCTIONS

Mnem	Name	Function
DIC	Read Status	Places the burst multiplexor channel flags in an accumulator.
DOA	Specify Low Order Address	Specifies the low order part of a memory address for loading or reading the first map register.
DOB*	Specify High Order Address	Specifies the high order part of a memory address for loading or reading the first map register.
DOB*	Specify Initial Map Register	Specifies the first map register of a group to be loaded or read.
DOC	Set Status	Used for diagnostic purposes only.

**These instructions are dependent on accumulator contents.*

MICROCODING

Writable Control Store

The writable control store (WCS) allows the user to transfer control to any one of 16 entry points in WCS. With these routines the full power of the microcode processor can be used.

Placing Microcode In WCS

Before the user can use the XOP feature to execute instructions in WCS, the microcode must be placed in the WCS locations. For a discussion of how to place microcode in WCS, see the *Load Control Store Formatted* instruction in Chapter V. For a detailed discussion of how to write microprograms see *Microprogramming ECLIPSE Computers with the WCS Feature (DGC No. 014-000620* and *Learning to Microprogram Your ECLIPSE Computer (DGC No. 014-000098).*

Writable Control Store Instructions

The following table describes the WCS instructions.

WCS INSTRUCTIONS

Mnem	Name	Function
LCSF	Load Control Store Formatted	Loads a number of microinstructions into WCS.
XOP1	Enter WCS	Executes a microprogram in WCS.

Integral Array Processor

Some computing applications, such as scientific programming, require operations that must be performed on each element of a large data array. General purpose computers solve this problem with program loops. However, this solution requires a group of perhaps several instructions that must be fetched and decoded each time the loop executes. The Integral Array Processor (IAP) has an instruction set that can perform a series of operations in one instruction, such as adding two arrays of numbers and storing the results in a third. The elimination of all the additional instructions results in simpler programs and a substantial improvement in speed.

Instruction Set

The IAP's instruction set is an extension of the standard ECLIPSE instruction set. These instructions range from simple functions such as array addition and subtraction up to digital filtering, signal processing, and Fourier transforms. Array-array and scalar-array arithmetic can be performed on 32-bit real or 64-bit complex numbers.

IAP instructions can be freely interspersed with the standard ECLIPSE instructions in a program. The IAP mnemonics are upward compatible with the standard ECLIPSE assembler mnemonics, so no additional software is needed by a user who intends to program the IAP in assembler language.

IAP instructions allow great flexibility to the user by employing the concept of a *parameter block*. This block consists of up to 16 words of main memory and contains all control information, such as array lengths and start addresses. The parameter block is never modified by any IAP instruction, so one block may be used repeatedly by more than one instruction.

The IAP maintains a number of 1-bit flags to detect error conditions, such as exponent overflow. These flags are placed in an accumulator at the end of each IAP instruction. They are also logically AND'ed with a mask word from the parameter block. If the result of the AND is nonzero, program control is transferred to the user's fault routine. This gives the user a high degree of control and reduces the extra programming involved in testing for errors.

Another powerful feature of the IAP is its *address step registers*. There are three of these 12-bit registers: two for source arrays and one for the result array. The contents of one of these registers is added to the current address before loading or storing each element of an array in memory. Thus, the IAP can reference every *n*th element in memory, where *n* is the value of the step register. Any positive, negative, or zero step value may be used. The step registers greatly simplify the programming of functions such as matrix transposition and data desampling.

Most IAP instructions have an option called the *partial parameter load*. Many applications involve manipulations on arrays where certain parameters do not change from one instruction to the next. A partial load instruction assumes the same attributes for these parameters as specified by a preceding instruction. This simplifies the programming and at the same time results in faster execution.

For more information on the IAP and its instruction set, refer to Appendix A of this manual.

Array Processing Instructions

Scalar Instructions

These six instructions perform arithmetic operations on one scalar and one array operand. The full parameter load version of each instruction, e.g. **ARS**, takes the scalar from the parameter block. The partial load version, e.g. **ARSP**, uses the value of the scalar that is already in the working register.

Source and result arrays may *overlap*; that is, part or all of one array may occupy the same memory locations as the other. The AP operates on an element by element basis, so you must be sure that no result data will overwrite source data before they are used.

MNEMONIC	NAME
ARS	Add real scalar to array
SRS	Subtract real scalar from array
MRS	Multiply real scalar by array
MCS	Multiply complex scalar by array
SPS	Signed product of real scalar and array
CMS	Compare real scalar to array

Array Instructions

The 17 array instructions include arithmetic operations on two arrays, and also single-array functions such as sum of elements and search for maximum or minimum element.

Source and result arrays may *overlap*; that is, part or all of one array may occupy the same memory locations as the other. The IAP operates on an element by element basis, so you must be sure that no result data will overwrite source data before they are used.

MNEMONIC	NAME
ARA	Add real arrays
SRA	Subtract real arrays
MRA	Multiply real arrays
MCA	Multiply complex arrays
NRA	Negate real array
SMA	Square magnitudes of complex array
SPR	Signed product of real arrays
SER	Sum of elements of real array
PER	Product of elements of real array
PEC	Product of elements of complex array
IPR	Inner product of real arrays
IPC	Inner product of complex arrays
EPR	Evaluate polynomial in real array
EPC	Evaluate polynomial in complex array
MXR	Maximum element of real array
MNR	Minimum element of real array
CMA	Compare arrays

Signal Processing Instructions

The IAP instruction set includes seven signal processing functions. Available operations include convolution and correlation as well as several types of digital filtering.

MNEMONIC	NAME
CONR	Convolution of real arrays
CONC	Convolution of complex arrays
CORR	Correlation of real arrays
CORC	Correlation of complex arrays
RFR	Recursive filter of real data
RFC	Recursive filter of complex data
INR	Integrate real array
FFTC	Fast Fourier transform of complex array
FFTR	Fast Fourier transform of real array
BRC	Bit-reverse indices of complex array
SCB	Store complex array bit-reversed

Use of FFT instructions

When computing the Fourier transform for a real array, you must use the **FFTC** and **FFTR** instructions in combination as summarized below.

The specific instruction sequence to use for a forward real FFT is shown below.

```
;FORWARD REAL FFT
        ELEF    2,PBLK      ;load AC2 with
                            ;parameter block address
        FFTC                ;perform initial calculations
        BRC                 ;rearrange intermediate data
        FFTR                ;then complete the transform
```

For an inverse real FFT, use the following instruction sequence:

```
;INVERSE REAL FFT
        ELEF    2,PBLK1     ;(we need two parameter blocks for
                            ; this example -- SEE NOTE 1)
        FFTR                ;do initial calculations
        ELEF    2,PBLK2     ;get address of other block
        FFTC                ;then complete the transform
        BRC                 ;and unscramble result (SEE NOTE 2)
```

NOTES:

(1) Two parameter blocks are used here because the bits in word 8 have a different format for each instruction. If you prefer, you may write a routine to modify the bits instead of using two blocks.

(2) You can use the **SCB** *instruction instead of* **BRC** *if you wish to move the result to main memory.*

Data movement and conversion instructions

The IAP instruction set includes 12 instructions which are used to perform functions such as type conversion and data movement.

MNEMONIC	NAME
FLL	Float and load (convert integer to real)
FXS	Fix and store (convert real to integer)
LSR	Load scratchpad registers
SSR	Store scratchpad registers
SRW	Store real scalar form working register
SCW	Store complex scalar from working register
LDR	Load real array (from main memory)
LDC	Load complex array (from main memory)
STR	Store real array (to main memory)
STC	Store complex array (to main memory)
CRE	Create a real array
MOD	Modify a real array

Satellite Processor

The Satellite Processor is a self-contained ECLIPSE CPU located in the main chassis of the host ECLIPSE S/250. Its function is to ease the load of the host by taking over such tasks as I/O device control, signal processing, and array processing. It executes an extended ECLIPSE instruction set, and contains 64 Kbytes of local memory plus all the processing features of the ECLIPSE. An S/250 system may contain up to eight SPs.

Instruction Set

The SP uses the standard ECLIPSE instruction set, plus floating point instructions and four special instructions used in host-SP communications. The standard instructions and the special communications instructions are microcoded. The floating point instructions use software routines that simulate floating point manipulation.

Host-SP Communication

Both the host and the SP use special instructions when communicating with one another. Since the host cannot directly access SP local memory, it must rely on its five special instructions to control the SP. These instructions act much as the front panel of a processor does: they can be used to load data in the SP memory, check accumulators or locations, or check the result of the last operation that took place. The SP can access host memory directly; its four special instructions specify the location of data, control the SP timer, or check on map status.

SP Data Channel

The SP maintains its own data channel separate from that of the host. The SP MAP uses this SP data channel to make data transfers to SP I/O devices. These data transfers require little interference from the host, thereby saving time.

SP MAP

The SP MAP can map 2 Kbyte pages of SP address space into either SP local memory or host memory. The MAP intercepts memory references and extracts a 1-bit pointer from the 15-bit address. The pointer determines whether the address is to be located in local or host memory. Addresses to be located in SP local memory are unchanged by the MAP. Addresses to be located in host memory are sent across the host data channel to the host MAP. The host MAP interprets the physical location of the address sent by the MAP.

The 8661 SP

For situations where large amounts of data must be processed in a short time, an SP containing an array processor is available. This arrangement, called an 8661 SP, can perform signal processing, array processing, or other similar tasks. The 8661 SP contains 56 Kbytes of local SP memory and another 8 Kbytes of AP memory. The 8661 SP can be combined with other SPs in any combination, up to the eight total units for one host processor.

The 8661 SP executes all SP instructions as well as AP instructions. For a discussion of the AP instruction set, refer to the section on the Integral Array Processor in this chapter.

For more detailed information about host-SP communications, refer to Appendix A in the back of this manual.

SP Instructions

The SP recognizes the standard ECLIPSE instruction set. Since these instructions are described elsewhere in this manual, we merely list the names and the mnemonics of the instructions here.

SP Instructions

MNEMONIC	NAME
ADC	Add Complement
ADD	Add
ADDI	Extended Add Immediate
ADI	Add Immediate
ANC	AND with Complemented Source
AND	Logical AND
ANDI	AND Immediate
BAM	Block Add and Move
BLM	Block Move
BTO	Set Bit to One
BTZ	Set Bit to Zero
CLM	Compare to Limits
COB	Count Bits
COM	Complement
DAD	Decimal Add
DHXL	Double Hex Shift Left
DHXR	Double Hex Shift Right
DIV	Unsigned Divide
DIVS	Signed Divide
DIVX	Sign Extend and Divide
DLSH	Double Logical Shift
DSB	Decimal Subtract
DSZ	Decrement and Skip if Zero
EDSZ	Extended Decrement and Skip if Zero
EISZ	Extended Increment and Skip if Zero
EJMP	Extended Jump
EJSR	Extended Jump to Subroutine
ELDA	Extended Load Accumulator
ELDB	Extended Load Byte
ELEF	Extended Load Effective Address
ESTA	Extended Store Accumulator
ESTB	Extended Store Byte
HLV	Halve
HXL	Hex Shift Left
HXR	Hex Shift Right
INC	Increment
IOR	Inclusive OR
IORI	Exclusive OR Immediate
ISZ	Increment and Skip if Zero
JMP	Jump
JSR	Jump to Subroutine
LDA	Load Accumulator
LDB	Load Byte
LMP	Load Map
LOB	Locate Lead Bit
LSH	Logical Shift
LSN	Load Sign
MOV	Move
MSP	Modify Stack Pointer

SP Instructions Cont.

MNEMONIC	NAME
MUL	Unsigned Multiply
MULS	Signed Multiply
NEG	Negate
POP	Pop Multiple Accumulators
POPB	Pop Block
POPJ	Pop PC and Jump
PSH	Push Multiple Accumulators
PSHJ	Push Jump
PSHR	Push Return Address
RSTR	Restore
RTN	Return
SAVE	Save
SBI	Subtract Immediate
SGE	Skip if ACS Greater Than or Equal to ACD
SGT	Skip if ACS Greater Than ACD
SNB	Skip on Non-zero Bit
STA	Store Accumulator
STB	Store Byte
SUB	Subtract
SYC*	System Call
SZB	Skip on Zero Bit
SZBO	Skip on Zero Bit and Set to One
XCH	Exchange Accumulators
XCT	Execute
XOP	Extended Operation
XOP1	Enter WCS
XOR	Exclusive OR
XORI	Exclusive OR Immediate

*The SYC instruction does not disable the MAP on the SP.

The 8660 SP also recognizes the Character Instruction Set:

MNEMONIC	NAME
CMP	Character Compare
CMT	Character Move Until True
CMV	Character Move
CTR	Character Translate

The SP recognizes the following I/O instructions

MNEMONIC	NAME
DIA	Data In A
DIB	Data In B
DIC	Data In C
DOA	Data Out A
DOB	Data Out B
DOC	Data Out C
HALT*	Halt
INTA*	Interrupt Acknowledge
INTDS*	Interrupt Disable
INTEN*	Enable Interrupts
IORST*	I/O Reset
MSKO*	Mask Out
NIO	No I/O Transfer
READS*	Read Switches
SKP	I/O Skip
VCT	Vector On Interrupting Device Code

*The SP also recognizes the general form of these instructions.

Chapter IV
ECLIPSE S/250 INSTRUCTIONS

This chapter lists all the instructions for the machine *except* I/O instructions and those intended for a specific device such as the MAP, the BMC, and the CPU. We have arranged the instructions in alphabetical order according to mnemonics as recognized by the assembler.

For each instruction we include:

- the mnemonic recognized by the assembler
- the bit format required
- the format of any arguments involved
- a functional description of each instruction

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
i	Signed two's complement integer in the range -32,768 to 32,767; or unsigned in the range 0 to 65,535
N	Integer in the range 0-3
n	Integer in the range 1-4
AC	Accumulator
ACS	Source accumulator
ACD	Destination accumulator
FPAC	Floating point accumulator
FACS	Floating point source accumulator
FACD	Floating point destination accumulator

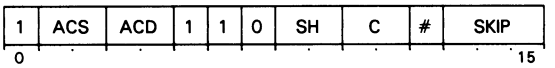
Setting the Index Field

To set the index field, code a comma followed by an integer between 0-3. This will set the index field to the value you specified. You can also use the symbol **dot** (.) to set the index field to 01. **Dot** can be read as the **address of the instruction.** When you use **dot,** you usually follow it with a plus or minus sign and the displacement value, such as **.+3** or **.-12.**

If you are coding extended class instructions, note that using a dot (e.g., **EJMP .+5** does not produce the same effect as coding a comma followed by a 1 (**EJMP 5,1**). When using a dot, the displacement is added to the address of the instruction (the first word of a two-word instruction). When using a comma, the displacement is added to the address of the word containing the displacement (the second word of a two-word instruction). Therefore, **EJMP .+5** is equivalent to **EJMP 4,1.**

Add

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



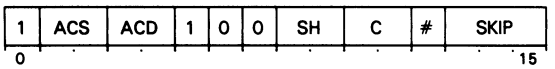
Performs unsigned integer addition and complements carry if appropriate.

Initializes carry 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, carry 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 carry.*

Add Complement

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



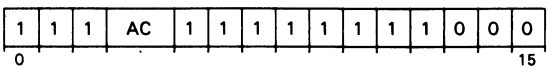
Adds the logical complement of an unsigned integer to another unsigned integer.

Initializes carry 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, carry 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 carry.*

Extended Add Immediate

ADDI *i,ac*

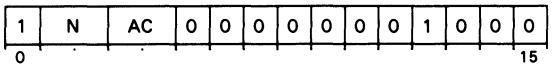


Adds a signed integer in the range -32,768 to +32,767 to the contents of an accumulator.

Treats the contents of the immediate field as a signed, 16-bit, two's complement number and adds it to the signed, 16-bit, two's complement number contained in the specified accumulator, placing the result in the same accumulator. Carry remains unchanged.

Add Immediate

ADI *n,ac*

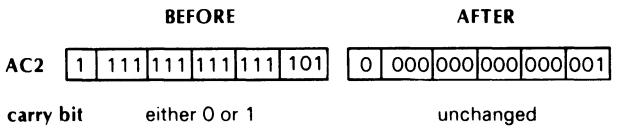


Adds an unsigned integer in the range 1-4 to the contents of an accumulator.

Adds the contents of the immediate field *N*, plus 1, to the unsigned, 16-bit number contained in the specified accumulator, placing the result in the same accumulator. Carry remains unchanged.

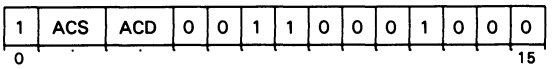
> NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, you should code the exact value that you wish to add.*

Example - Assume that AC2 contains 177775_8. After the instruction **ADI 4,2** is executed, AC2 contains 000001_8 and carry is unchanged.



AND With Complemented Source

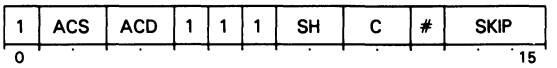
ANC *acs,acd*



Forms the logical AND of the logical complement of the contents of ACS and the contents of ACD and places the result in ACD. The instruction sets a bit position in the result to 1 if the corresponding bit position in ACS contains 0. The contents of ACS remain unchanged.

AND

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

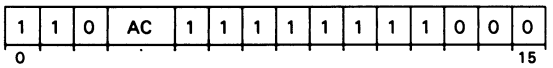


Forms the logical AND of the contents of two accumulators.

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.

AND Immediate

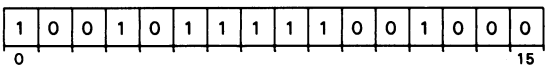
ANDI *i,ac*



Places the logical AND of the contents of the immediate field and the contents of the specified accumulator in the specified accumulator.

Block Add and Move

BAM



Moves memory words from one location to another, adding a constant to each one.

Moves words sequentially from one memory location to another, treating them as unsigned, 16-bit integers. After fetching a word from the source location, the instruction adds the unsigned, 16-bit integer in AC0 to it. If the addition produces a carry of 1 out of the high-order bit, no indication is given.

Bits 1-15 of AC2 contain the address of the source location. Bits 1-15 of AC3 contain the address of the destination location. The address in bits 1-15 of AC2 or AC3 is an indirect address if bit 0 of that accumulator is 1. In that case, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

The unsigned, 16-bit number in AC1 is equal to the number of words moved. This number must be greater than 0 and less than or equal to 32,768. If the number in AC1 is outside these bounds, no data is moved and the contents of the accumulators remain unchanged.

AC	CONTENTS
0	Addend
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the count in AC1 is decremented by one and the source and destination addresses in AC2 and AC3 are incremented by one. Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following the last word in their respective fields. The contents of AC0 remain unchanged.

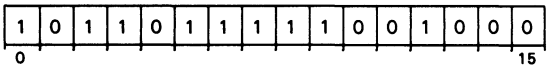
Words are moved in consecutive, ascending order according to their addresses. The next address after 77777_8 is 0 for both fields. The fields may overlap in any way.

NOTE: *Because of the potentially long time that may be required to perform this instruction it is interruptable. If a* Block Add and Move *instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the interrupted instruction. Because the addresses and the word count are updated after every word stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the* Block Add and Move *instruction.*

When updating the source and destination addresses, the *Block Add And Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Block Add And Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Block Move

BLM



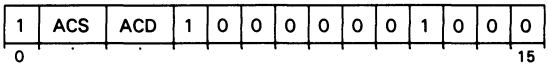
Moves memory words from one location to another.

The *Block Move* instruction is the same as the *Block Add And Move* instruction in all respects except that no addition is performed and AC0 is not used.

NOTE: *The* Block Move *instruction is interruptible in the same manner as the* Block Add And Move *instruction.*

Set Bit To One

BTO *acs,acd*



Sets the specified bit to 1.

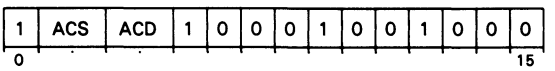
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16-bits of the bit pointer and assumes the high-order 16 bits are 0.

The instruction then sets the addressed bit in memory to 1, leaving the contents of ACS and ACD unchanged.

NOTE: *The bit pointer contained in ACS and ACD must not make indirect memory references.*

Set Bit To Zero

BTZ *acs,acd*



Sets the addressed bit to 0.

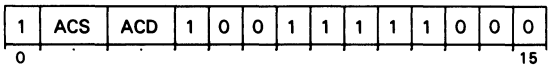
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

The instruction then sets the addressed bit in memory to 0, leaving the contents of ACS and ACD unchanged.

NOTE: *The bit pointer contained in ACS and ACD must not make indirect memory references.*

Compare To Limits

CLM *acs,acd*



Compares a signed integer with two other integers and skips if the first integer is between the other two. The accumulators determine the location of the three integers.

Compares the signed, two's complement integer in ACS to two signed, two's complement limit values, *L* and *H*. If the number in ACS is greater than or equal to L and less than or equal to H, the next sequential word is skipped. If the number in ACS is less than L or greater than H, the next sequential word is executed.

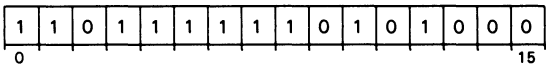
If ACS and ACD are specified as different accumulators, the address of the limit value L is contained in bits 1-15 of ACD. The limit value H is contained in the word following L. Bit 0 of ACD is ignored.

If ACS and ACD are specified as the same accumulator, then the integer to be compared must be in that AC and the limit values L and H must be in the two words following the instruction. L is the first word and H is the second word. The next sequential word is the third word following the instruction.

Character Compare

CMP

Optional Instruction (with CIS)



Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range ($0\text{-}255_{10}$). If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addressed) for each string.

AC0 specifies the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, ACO contains the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in string 2.

AC1 specifies the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in string 1.

AC2 contains a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

CODE	COMPARISON RESULT
- 1	string 1 < string 2
0	string 1 = string 2
+ 1	string 1 > string 2

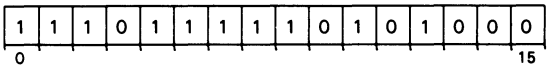
The strings may overlap in any way. Overlap will not effect the results of the comparison.

Upon completion, AC0 contains the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table above. AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality were found), or to the byte following string 2 (if string 2 were exhausted). AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality were found), or to the byte following string 1 (if string 1 were exhausted). If the length of both string 1 and string 2 were zero, the instruction returns 0 in AC1. If the two strings are of unequal length, the instruction pads the shorter string with space characters $<040_8>$ and continues the comparison.

Character Move Until True

CMT

Optional Instruction (with CIS)



Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0\text{-}255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter, and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it, and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*), or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

AC0 contains the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

AC1 specifies the length of the strings and the direction of processing. If the source string is to be moved to the destination field in ascending order, AC1 contains the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination field in descending order, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the process is performed in ascending order, AC2 points to the lowest byte in the destination field. When the process is performed in descending order, AC2 points to the highest byte in the destination field.

AC3 contains a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, AC3 points to the lowest byte in the source string. When the process is performed in descending order, AC3 points to the highest byte in the source string.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

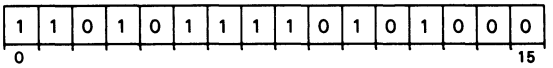
Upon completion, AC0 contains the resolved address of the translation table and AC1 contains the number of bytes that were not moved. AC2 contains a byte pointer to the byte following the last byte written in the destination field. AC3 contains a byte pointer either to the delimiter or to the first byte following the source string.

NOTE: *If AC1 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. The instruction becomes a No-Op.*

Character Move

CMV

Optional Instruction (with CIS)



Under control of the four accumulators, moves a string of bytes from one area of memory to another and returns a value in the Carry bit reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

AC0 specifies the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in the destination field.

AC1 specifies the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, AC1 contains the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, AC2 points to the lowest byte. When the field is written in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, AC3 points to the lowest byte. When the field is copied in descending order, AC3 points to the highest byte.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains 0 and AC1 contains the number of bytes left to fetch from the source field. AC2 contains a byte pointer to the byte following the destination field; and AC3 contains a byte pointer to the byte following the last byte fetched from the source field.

NOTE: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters.*

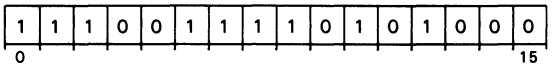
If the source field is longer than the destination field, the instruction terminates when the destination field is filled and sets carry to 1. In any other case, the instruction sets carry to 0.

If the source field is shorter than the destination field, the instruction pads the destination field with space characters $<040_8>$.

Character Translate

CTR

Optional Instruction (with CIS)



Under control of the four accumulators, translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

The instruction operates in two modes; translate and move, and translate and compare. When operating in translate and move mode, the instruction translates each byte in string 1, and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.

When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above, and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range ($0\text{-}255_{10}$). If two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition the *lower valued* string. Both strings remain unchanged.

ACO specifies the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

AC1 specifies the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, AC1 contains the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, AC1 contains the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

AC2 contains a byte pointer to the first byte in string 2.

AC3 contains a byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, AC0 contains the address of the word which contains the byte pointer to the translation table and AC1 contains 0. AC2 contains a byte pointer to the byte following string 2 and AC3 contains a byte pointer to the byte following string 1.

Upon completion of a translate and compare operation, AC0 contains the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the table below. AC2 contains a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 if the strings were identical. AC3 contains a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical.

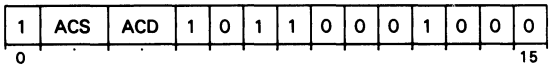
CODE	RESULT
-1	Translated value of string 1 < Translated value of string 2
0	Translated value of string 1 = Translated value of string 2
+1	Translated value of string 1 > Translated value of string 2

If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

Count Bits

COB *acs,acd*

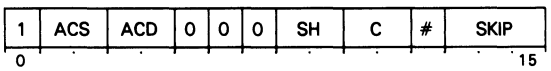


Adds a number equal to the number of ones in ACS to the signed, 16-bit, two's complement number in ACD. The instruction leaves the contents of ACS and the state of carry unchanged.

NOTE: *If ACS and ACD are the same accumulator, the instruction functions as described above, except the contents of ACS will be changed.*

Complement

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

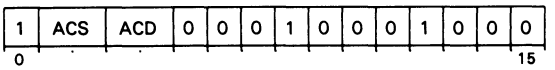


Forms the logical complement of the contents of an accumulator.

Initializes carry 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.

Decimal Add

DAD *acs,acd*



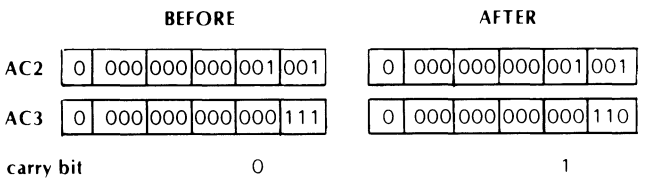
Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses the carry bit for a decimal carry.

Adds the unsigned decimal digit contained in ACS bits 12-15 to the unsigned decimal digit contained in ACD bits 12-15. The carry bit is added to this result. The instruction then places the decimal units' position of the final result in ACD bits 12-15, and the decimal carry in the carry bit. The contents of ACS and bits 0-11 of ACD remain unchanged.

NOTE: *No validation of the input digits is performed. Therefore, if bits 12-15 of either ACS or ACD contain a number greater than 9, the results will be unpredictable.*

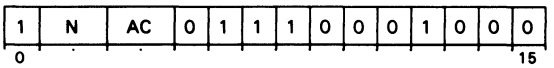
Example:

Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0. After the instruction **DAD 2,3** is executed, AC2 remains the same; bits 12-15 of AC3 contain 6; and the carry bit is 1, indicating a decimal carry from this *Decimal Add*.



Double Hex Shift Left

DHXL *n,ac*



Shifts the 32-bit number contained in AC and AC+1 left a number of hex digits depending upon the immediate field N. The number of digits shifted is equal to N+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes.

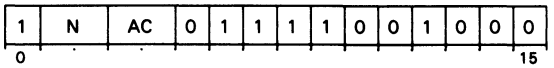
NOTE: *If AC is specified as AC3, then AC+1 is AC0.*

The assembler takes the coded value of n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

If N is equal to 3, the contents of AC+1 are placed in AC and AC+1 is filled with zeroes.

Double Hex Shift Right

DHXR *n,ac*



Shifts the 32-bit number contained in AC and AC+1 right a number of hex digits depending upon the immediate field N. The number of digits shifted is equal to N+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes.

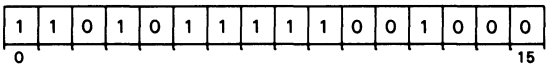
NOTE: *If AC is specified as AC3, then AC+1 is AC0.*

The assembler takes the coded value of n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

If N is equal to 3, the contents of AC are placed in AC+1 and AC is filled with zeroes.

Unsigned Divide

DIV



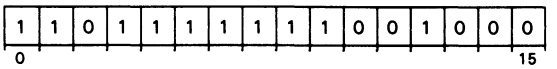
Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

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. Carry 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. Carry is set to 1, and the operation is terminated. All operands remain unchanged.*

Signed Divide

DIVS



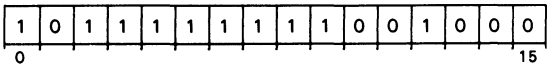
Divides the signed 32-bit integer in two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

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. Carry 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. Carry is set to 1, and the operation is terminated. The contents of AC0 and AC1 are unpredictable.*

Sign Extend and Divide

DIVX

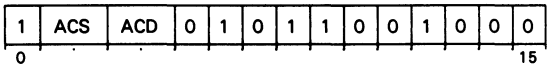


Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.

Extends the sign of the number in AC1 into AC0 by placing a copy of bit 0 of AC1 in each bit of AC0. After extending the sign, the instruction performs a *Signed Divide* operation.

Double Logical Shift

DLSH *acs,acd*



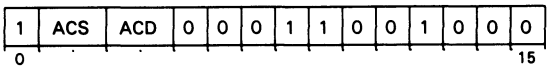
Shifts the 32-bit number contained in ACD and ACD+1 either left or right depending on the number contained in bits 8-15 of ACS. The signed, 8-bit two's complement number contained in bits 8-15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8-15 of ACS is positive, shifting is to the left; if the number in bits 8-15 of ACS is negative, shifting is to the right. If the number in bits 8-15 of ACS is zero, no shifting is performed. Bits 0-7 of ACS are ignored.

AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 8-15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged.

NOTE: *If the magnitude of the number in bits 8-15 of ACS is greater than 31_{10}, all bits of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Decimal Subtract

DSB *acs,acd*

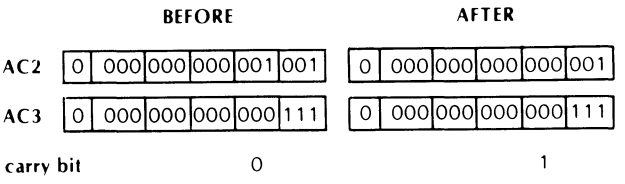


Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses carry as a decimal borrow.

Subtracts the unsigned decimal digit contained in ACS bits 12-15 from the unsigned decimal digit contained in ACD bits 12-15. Subtracts the complement of carry from this result. Places the decimal units' position of the final result in ACD bits 12-15 and the complement of the decimal borrow in carry. In other words, if the final result is negative, the instruction indicates a borrow and sets carry to 0. If the final result is positive, the instruction indicates no borrow and sets carry to 1. The contents of ACS and bits 0-11 of ACD remain unchanged.

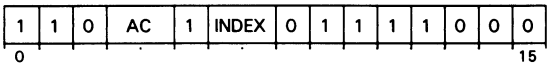
Example:

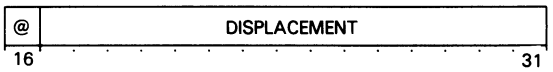
Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and carry contains 0. After the instruction **DSB 3,2** is executed, AC3 remains the same; bits 12-15 of AC2 contain 1; and carry is set to 1, indicating no borrow from this *Decimal Subtract*.



Dispatch

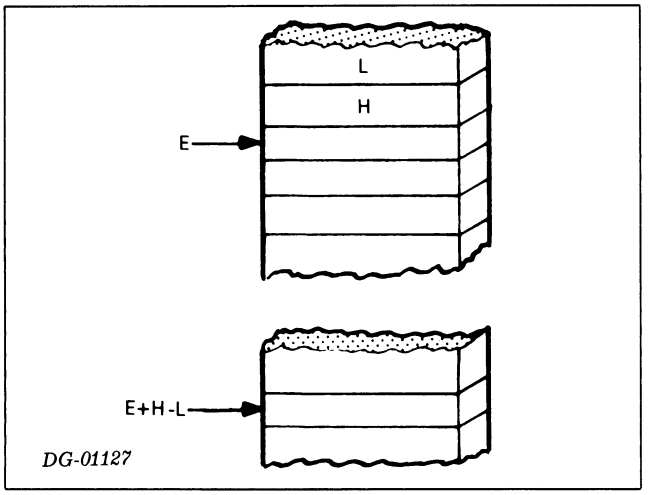
DSPA *ac,[@]displacement[,index]*





Conditionally transfers control to an address selected from a table.

Computes the effective address *E*. This is the address of a *dispatch table*. The dispatch table consists of a table of addresses. Immediately before the table are two signed, two's complement limit words, *L* and *H*. The last word of the table is in location E+H-L.

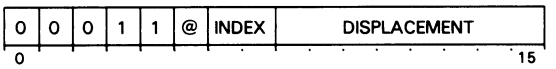


Compares the signed, two's complement number contained in the accumulator to the limit words. If the number in the accumulator is less than *L* or greater than *H*, sequential operation continues with the instruction immediately after the *Dispatch* instruction.

If the number in AC is greater than or equal to *L* and less than or equal to *H*, the instruction fetches the word at location E-L+number. If the fetched word is equal to 177777_8, sequential operation continues with the instruction immediately after the *Dispatch* instruction. If the fetched word is not equal to 177777_8, the instruction treats this word as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter and sequential operation continues with the word addressed by the updated value of the program counter.

Decrement And Skip If Zero

DSZ *[@]displacement[,index]*

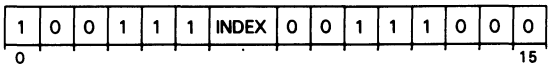


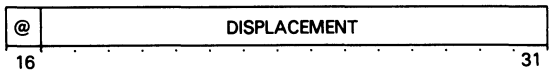
Decrements the addressed word, then skips if the decremented value is zero.

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.

Extended Decrement and Skip if Zero

EDSZ *[@]displacement[,index]*



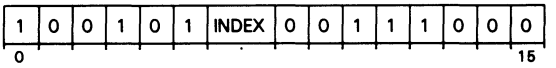


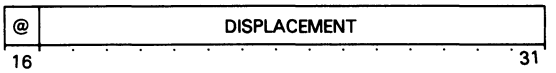
Decrements the addressed word, then skips if the decremented value is zero.

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

Extended Increment And Skip If Zero

EISZ *[@]displacement[,index]*



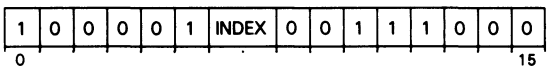


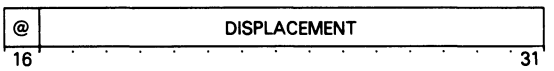
Increments the addressed word, then skips if the incremented value is zero.

Computes the effective address, *E*. Increments by one the contents of the location specified by *E*, and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction skips the next sequential word.

Extended Jump

EJMP *[@]displacement[,index]*

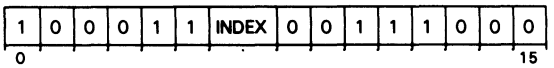


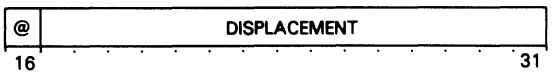


Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Extended Jump To Subroutine

EJSR *[@]displacement[,index]*





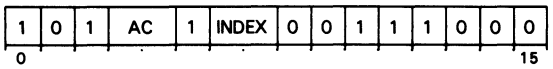
Increments and stores the value of the program counter in AC3, then places a new address in the program counter.

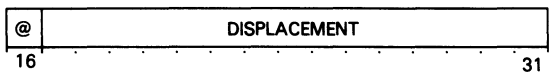
Computes the effective address, *E*. The instruction then places the address of the next sequential instruction (the instruction following the *EJSR* instruction) in AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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

Extended Load Accumulator

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



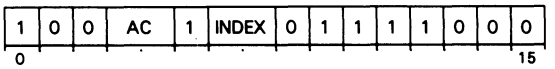


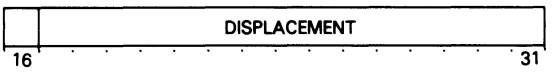
Moves a copy of the contents of a memory word into the specified accumulator.

Calculates the effective address, *E*. Places the contents of the location addressed by *E* in the specified accumulator. The contents of the location addressed by *E* remain unchanged.

Extended Load Byte

ELDB *ac,displacement[,index]*





Copies a byte from memory into an accumulator.

Forms a byte pointer from the displacement in the following way: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement to give a memory address. The byte indicator designates which byte of the addressed word will be loaded into bits 8-15 of the specified accumulator. The instruction sets bits 0-7 of the specified accumulator to 0.

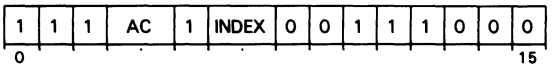
The instruction destroys the previous contents of the specified accumulator, but it does not alter either the index value or the displacement.

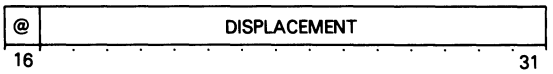
The argument *index* selects the source of the index value. It may have values in the range of 0-3. The meaning of each value is shown below:

INDEX BITS	INDEX VALUE
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Load Effective Address

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





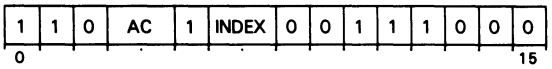
Places an effective address in an accumulator.

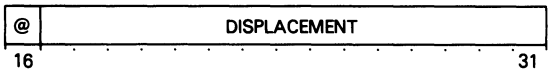
Computes the effective address, *E*, and places it in bits 1-15 of the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the accumulator are lost.

```
ELEF   0,TABLE      ; The logical address of TABLE
                    ; is placed in AC0.
ELEF   1,-55,3      ; Subtracts 000055 (octal) from
                    ; the unsigned integer in AC3 and
                    ; places the result in AC1.
ELEF   0,.+0        ; Places the logical address of this
                    ; Load effective address
                    ; instruction in AC0.
```

Extended Store Accumulator

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



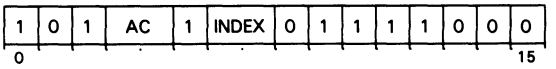


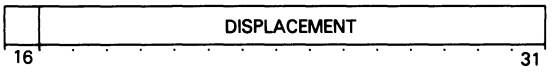
Stores the contents of an accumulator into a memory location.

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

Extended Store Byte

ESTB *ac,displacement[,index]*





Copies into memory the byte contained in the right half of an accumulator.

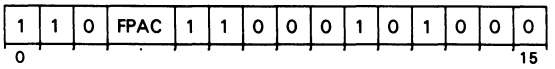
Forms a byte pointer from the displacement as follows: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement field to give a memory address. The byte indicator determines which byte of the addressed location will receive bits 8-15 of the specified accumulator.

The argument *index* selects the source of the index value. It may have values in the range of 0-3; the meaning of each value is shown below:

INDEX BITS	INDEX VALUE
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Absolute Value

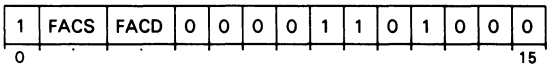
FAB *fpac*



Sets the sign bit of FPAC to 0. Also sets the exponent to zero if the mantissa is zero; otherwise leaves bits 1-63 of FPAC unchanged. Updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

Add Double (FPAC to FPAC)

FAD *facs,facd*



Adds the floating point number in FACS to the floating point number in FACD and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FACD.

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. One guard digit is provided so that all but four 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 15 hex digits.

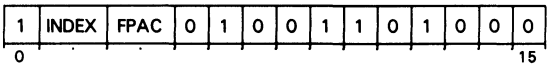
After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by 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 floating point status register, and the number in FACD is correct, except that the exponent is 128 too small.

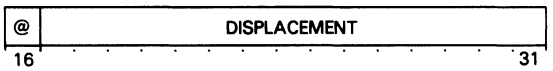
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FACD and the instruction terminates.

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

Add Double (Memory to FPAC)

FAMD *fpac,[@]displacement[,index]*





Adds the floating point number in the source location to the floating point number in FPAC and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged and updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address *E* which addresses a 4-word (double precision) operand.

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. One guard digit is provided so that all but four 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 15 hex digits for double precision, or 7 hex digits for single precision.

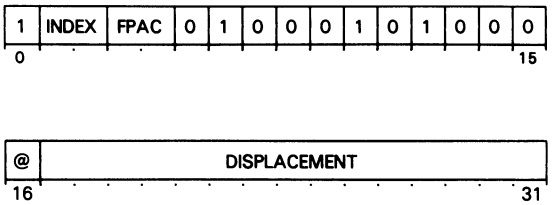
After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by 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 floating point status register, and 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 the FPAC and the instruction terminates.

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

Add Single (Memory to FPAC)

FAMS *fpac,[@]displacement[,index]*



Adds the floating point number in the source location to the floating point number in FPAC and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged and updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address, *E*, which addresses a 2-word (single precision) operand.

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. One guard digit is provided so that all but four 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 15 hex digits for double precision, or 7 hex digits for single precision.

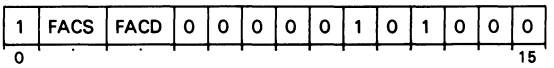
After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by 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 floating point status register, and 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 the FPAC and the instruction terminates.

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

Add Single (FPAC to FPAC)

FAS *facs,facd*



Adds the floating point number in FACS to the floating point number in FACD and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged and updates the Z and N flags in the floating point status register to reflect the new contents of FACD.

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. One guard digit is provided so that all but four 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 15 hex digits for double precision, or 7 hex digits for single precision.

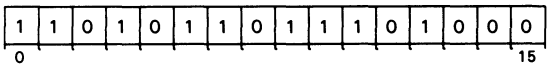
After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by 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 floating point status register, and the number in FACD 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 the FACD and the instruction is terminated.

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

Clear Errors

FCLE

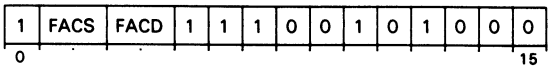


Sets bits 0-4 of the floating point status register to 0.

NOTE: *The I/O RESET instruction will also set these bits to 0.*

Compare Floating Point

FCMP *facs,facd*



Compares two floating point numbers and sets the Z and N flags in the floating point status register accordingly.

Algebraically compares the floating point numbers in FACS and FACD to each other and updates the Z and N flags in the floating point status register to reflect the result. Leaves the contents of FACS and FACD unchanged. The results of the compare and the corresponding flag settings are shown in the table below.

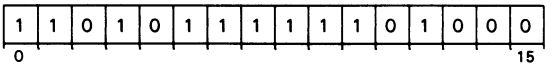
Z	N	RESULT
1	0	FACS=FACD
0	1	FACS>FACD
0	0	FACS<FACD

NOTE: *Unnormalized operands give unspecified results.*

Cosine Double

FCOSD

Optional Instruction (with FPP)



Forms the cosine of the number in FPAC0, places the result in FPAC0, and sets the Z and N flags of the floating point status register to reflect the new value in FPAC0. Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

The *Sine* and *Cosine* instructions can share the same data. The *Sine* instruction always skips the word immediately following the instruction word when searching for data. The *Cosine* instruction word can be placed in this location if desired.

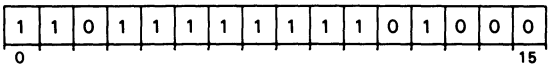
Format: Algorithm coefficients must follow the *Cosine* instruction. The format is:

WORD	NAME	CODED VALUE (Hex)			
0	Instruction Word	FCOSD			
1-4	4/PI	4114	5F30	6DC9	C883
5-8	A6	387C	F24A	053B	3668
9-12	A5	BA69	B262	61F8	B3A0
13-16	A4	3C3C	3E9F	5C1F	7D86
17-20	A3	BE15	5D3C	7DB7	837F
21-24	A2	3F40	F07C	206B	FE84
25-28	A1	C04E	F4F3	26F9	15EC
29-32	A0	40FF	FFFF	FFFF	FFCC
33-36	B6	3778	FBB4	E1B7	2DE0
37-40	B5	B978	C018	E66C	04DB
41-44	B4	3B54	1E0B	F28C	7BD1
45-48	B3	BD26	5A59	9C5A	A5E8
49-52	B2	3EA3	35E3	3BAC	37D9
53-56	B1	C014	ABBC	E625	BE3C
57-60	B0	40C9	0FDA	A221	6896

Cosine Single

FCOSS

Optional Instruction (with FPP)



Forms the cosine of the number in FPAC0, places the result in FPAC0, and sets the Z and N flags of the floating point status register to reflect the new value in FPAC0. Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

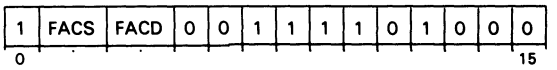
The *Sine* and *Cosine* instructions can share the same data. The *Sine* instruction always skips the word immediately following the instruction word when searching for data. The *Cosine* instruction word can be placed in this location if desired.

Format: Algorithm coefficients must follow the *Cosine* instruction. The format is:

WORD	NAME	CODED VALUE (Hex)	
0	Instruction Word	FCOSS	
1-2	4/PI	4114	5F30
3-4	A3	BE14	E35E
5-6	A2	3F40	EBCA
7-8	A1	C04E	F4E3
9-10	A0	40FF	FFFF
11-12	B3	BD25	B25F
13-14	B2	3EA3	2F49
15-16	B1	C014	ABBC
17-18	B0	40C9	0FDB

Divide Double (FPAC by FPAC)

FDD *facs,facd*



Divides the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FACD.

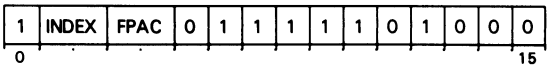
The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the floating point status register and the instruction is terminated. The number in FACD remains unchanged. If the mantissa is nonzero, the previous contents of FACD are lost. The two mantissas are compared and if the mantissa of the number in FACD is greater than or equal to the mantissa of the source operand, the mantissa of the number in FACD is shifted right one hex digit and the exponent of the number in FACD is increased by one. This process continues until the mantissa of the number in FACD is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

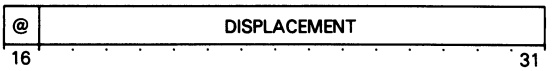
The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The 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 FACD.

If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FACD is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Divide Double (FPAC by Memory)

FDMD *fpac,[@]displacement[,index]*





Divides the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address, *E*, which addresses a 4-word (double precision) operand.

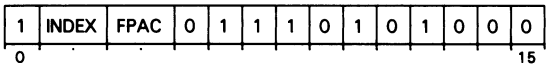
The source operand is checked for a zero mantissa. If the mantissa is zero, the **DVZ** bit is set in the floating point status register and the instruction is terminated. The number in FPAC remains unchanged. If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared 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. This process continues until the mantissa of the number in FACD is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

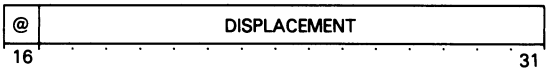
The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The 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 corresponding bit in the floating point status register is set. 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.

Divide Single (FPAC by Memory)

FDMS *fpac,[@]displacement[,index]*





Divides the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the **Z** and **N** flags in the floating point status register to reflect the new contents of FPAC.

Computes the effective address *E* which addresses a 2-word (single precision) operand.

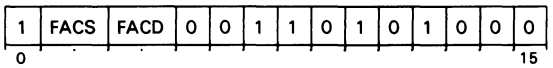
The source operand is checked for a zero mantissa. If the mantissa is zero, the **DVZ** bit is set in the floating point status register and the instruction is terminated. The number in FPAC remains unchanged. If the mantissa is nonzero, the previous contents of FPAC are lost. The two mantissas are compared 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. This process continues until the mantissa of the number in FPAC is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

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 corresponding bit in the floating point status register is set. 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.

Divide Single (FPAC by FPAC)

FDS *facs,facd*



Divides the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FACD.

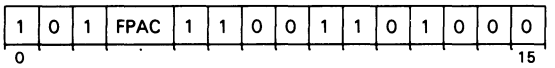
The source operand is checked for a zero mantissa. If the mantissa is zero, the DVZ bit is set in the floating point status register and the instruction is terminated. The number in FACD remains unchanged. If the mantissa is nonzero, the previous contents of FACD are lost. The two mantissas are compared, and if the mantissa of the number in FACD is greater than or equal to the mantissa of the source operand, the mantissa of the number in FACD is shifted right one hex digit and the exponent of the number in FACD is increased by one. This process continues until the mantissa of the number in FACD is less than the mantissa of the source operand. Since one guard digit is provided, all but four bits shifted out are lost.

The mantissa in FACD 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 FACD 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 FACD.

If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FACD is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Load Exponent

FEXP *fpac*



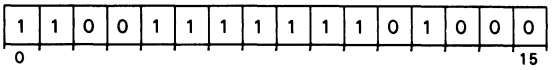
Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC. Ignores bits 0 and 8-15 of AC0. Leaves unchanged bits 0 and 8-63 of FPAC and the entire contents of AC0. Also sets bits 0-7 (the sign and exponent) to zero if bits 8-63 (the mantissa) of FPAC are zero. Leaves bits 1-7 of FPAC unchanged if FPAC contains true zero.

NOTE: *The exponent contained in bits 1-7 of AC0 is assumed to be in* Excess 64 *representation.*

Real Exponential Double

FEXPD

Optional Instruction (with FPP)



Raises the value of the mathematical constant *e*, to the power of the value in FPAC0, and places the result in FPAC0. Sets the Z and N flags of the floating point status register to reflect the new value in FPAC0.

Normal return: Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

Error return: Occurs if the exponential to be loaded into FPAC0 will cause an overflow or underflow (i.e.:

ABS (FPAC0) >= Ln (16^{63}) = 174.673

before the operation). Leaves AC3 and FPAC0 unchanged, and loads the address of the error return word into the program counter.

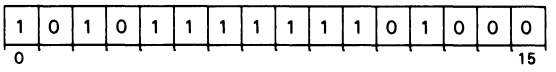
Format: Algorithm coefficients must follow the *Real Exponential* instructions . The standard format is:

WORD	NAME	CODED VALUE (Hex)			
0	Instruction Word	FEXPD			
1-4	LOGE	4117	1547	652B	82F9
5-8	LIMIT	C2AE	AC4F	97F2	880E
9-12	A2	3F5E	9721	55B8	5ED5
13-16	A1	4214	33BA	9313	EC1B
17-20	A0	435E	9E82	3FB9	A6DF
21-24	B1	42E9	2F28	7AE8	9543
25-28	B0	4411	1036	2F87	4CA5
29-32	SQ2X1	4116	A09E	667F	3BCD
33-36	SQ2X2	412D	413C	CCFE	7799
37-40	SQ2X4	415A	8279	99FC	EF33
41-44	SQ2X8	41B5	04F3	33F9	DE64
45	Error Address	(ADDR)			

Real Exponential Single

FEXPS

Optional Instruction (with FPP)



Raises the value of the mathematical constant *e*, to the power of the value in FPAC0, and places the result in FPAC0. Sets the Z and N flags of the floating point status register to reflect the new value in FPAC0.

Normal return: Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

Error return: Occurs if the exponential to be loaded into FPAC0 will cause an overflow or underflow, (i.e.:

ABS (FPAC0) >= Ln (16^{63}) = 174.673

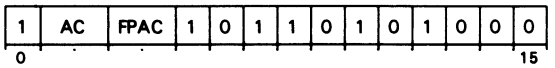
before the operation). Leaves AC3 and FPAC0 unchanged, and loads the address of the error return word into the program counter.

Format: Algorithm coefficients must follow the *Real Exponential* instructions . The standard format is:

WORD	NAME	CODED VALUE (Hex)	
0	Instruction Word	FEXPS	
1-2	LOGE	4117	1547
3-4	LIMIT	C2AE	AC4F
5-6	B	4219	0A03
7-8	A	418A	D86E
9-10	SQ2X1	4116	A09E
11-12	SQ2X2	412D	413D
13-14	SQ2X4	415A	827A
15-16	SQ2X8	41B5	04F3
17	Error Address	(ADDR)	

Fix To AC

FFAS *ac,fpac*



Converts the integer portion of the floating point number contained in the specified FPAC to a signed two's complement integer and places the result in an accumulator.

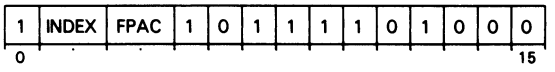
Forms the absolute value of the integer portion of the floating point number in FPAC. Extracts the 15 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result in the specified accumulator, sets the Z and N flags in the floating point status register to 0, and leaves the contents of FPAC unchanged.

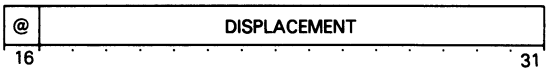
If the number in FPAC is less than -32,767 or greater than +32,767, this instruction sets the MOF flag in the floating point status register to 1.

NOTE: *If the lower 15 bits of the integer formed from the number in FPAC are all 0, the sign bit of the result will be zero, regardless of the sign of the original number.*

Fix To Memory

FFMD *fpac,[@]displacement[,index]*





Converts the integer portion of a floating point number to double-precision integer format and stores the result in two memory locations.

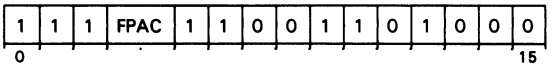
Forms the absolute value of the integer portion of the floating point number in FPAC. Extracts the 31 least significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result into the locations addressed by E, sets the Z and N flags in the floating point status register to 0, and leaves the contents of FPAC unchanged.

If the number in FPAC is less than -2,147,483,647 or greater than +2,147,483,647, this instruction sets the MOF flag in the floating point status register to 1.

If the lower 31 bits of the integer formed from the number in FPAC are all 0, the sign bit of the result will be zero.

Halve

FHLV *fpac*

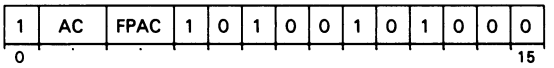


Divides the floating point number in FPAC by 2.

Shifts the mantissa contained in FPAC right one bit position, fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Then normalizes the number and places the result in FPAC. Sets the UNF flag in the floating point status register to 1 if the normalization process causes an exponent underflow. The number in FPAC is then correct, except that the exponent is 128 too large. Updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

Float From AC

FLAS *ac,fpac*



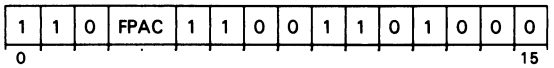
Converts a two's complement number to floating point format.

Converts the signed two's complement number contained in the specified accumulator to a single precision floating point number, places the result in the specified FPAC, and sets the low-order 32 bits of the FPAC to 0. Leaves the contents of the specified accumulator unchanged and destroys the previous contents of the FPAC. Updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

The range of numbers that can be converted is -32,768 to +32,767.

Integerize

FINT *fpac*

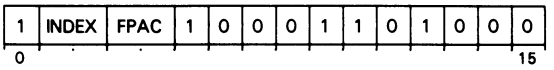


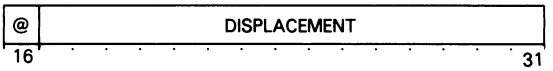
Zeros the fractional portion (if any) of the number contained in the specified FPAC, and then normalizes the number. The instruction updates the Z and N flags in the floating point status register to reflect the new contents of the specified FPAC.

> NOTE: *If the absolute value of the number contained in the specified FPAC is less than 1, the specified FPAC is set to true zero.*

Load Floating Point Double

FLDD *fpac,[@]displacement[,index]*



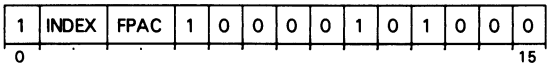


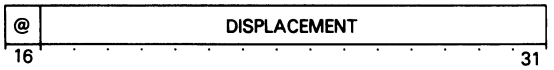
Moves four words out of memory into a specified FPAC.

Computes the effective address, *E*, and places the double precision floating point number at that address in FPAC. Also sets the sign and exponent to zero if the mantissa is zero. Destroys the previous contents of FPAC and updates the Z and N flags in the FPSR to reflect the new contents of FPAC.

Load Floating Point Single

FLDS *fpac,[@]displacement[,index]*



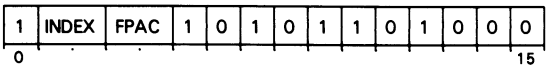


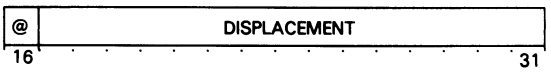
Moves two words out of memory into a specified FPAC.

Computes the effective address *E* and places the single precision floating point number at that address in FPAC. Also sets the sign and exponent to zero if the mantissa is zero. Destroys the previous contents of FPAC and updates the Z and N flags in the floating point status register to reflect the new contents of FPAC. The low-order 32 bits of FPAC are set to 0.

Float From Memory

FLMD *fpac,[@]displacement[,index]*





Converts the contents of two memory locations to floating point format and places the result in a specified FPAC.

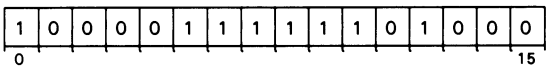
Computes the effective address *E*, converts the 32-bit, signed, two's complement number addressed by E to a double precision floating point number, and places the result in the specified FPAC. Destroys the previous contents of FPAC, and updates the Z and N flags in the floating point status register to reflect the new contents of the FPAC.

The range of numbers that can be converted is -2,147,483,648 to +2,147,483,647.

Natural Logarithm Double

FLOGD

Optional Instruction (with FPP)



Forms the natural logarithm of the floating point number in FPAC0 and puts the result into FPAC0. Sets the Z and N flags of the floating point status register to reflect the new contents of FPAC0.

Normal return: Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

Error return: Occurs if the original value in FPAC0 is less than or equal to zero (the logarithm function is invalid in this range). Leaves AC3 and FPAC0 unchanged, and loads the address of the error return word into the program counter.

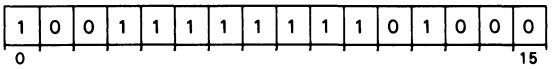
Format: Algorithm coefficients must follow the *Natural Logarithm* instructions. The format of the instruction is:

WORD	NAME	CODED VALUE (Hex)			
0	Instruction Word	FLOGD			
1-4	SQ.5	40B5	04F3	33F9	DE65
5-8	A2	C212	53EF	500D	FEAA
9-12	A1	425D	76C2	3149	ABB9
13-16	A0	C25A	2CB8	97BF	5916
17-20	B2	C214	BBC5	DCDB	3E86
21-24	B1	423D	C2D5	31EF	8B7F
25-28	B0	C22D	165C	4BDF	AC95
29-32	LOG2	40B1	7217	F7D1	CF7A
33	Error Address	(ADDR)			

Natural Logarithm Single

FLOGS

Optional Instruction (with FPP)



Forms the natural logarithm of the floating point number in FPAC0 and puts the result into FPAC0. Sets the Z and N flags of the floating point status register to reflect the new contents of FPAC0.

Normal return: Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

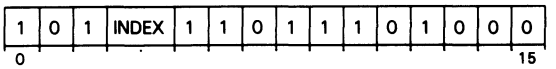
Error return: Occurs if the original value in FPAC0 is less than or equal to zero (the logarithm function is invalid in this range). Leaves AC3 and FPAC0 unchanged, and loads the address of the error return word into the program counter.

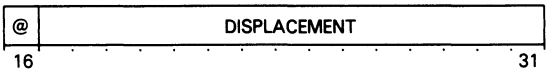
Format: Algorithm coefficients must follow the *Natural Logarithm* instructions. The format of the instruction is:

WORD	NAME	CODED VALUE (Hex)	
0	Instruction Word	FLOGS	
1-2	SQ.5	40B5	04F3
3-4	A1	40E5	4226
5-6	A0	C135	0453
7-8	B0	C11A	822A
9-10	LOG2	40B1	7218
11	Error Address	(ADDR)	

Load Floating Point Status

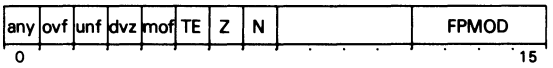
FLST *[@]displacement[,index]*

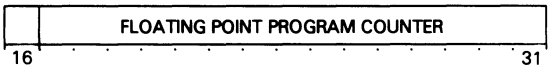




Moves the contents of two specified memory locations to the floating point status register.

Computes the effective address, *E*, places the 32-bit operand addressed by E in the floating point status register, and sets the condition codes to the values of the loaded bits.



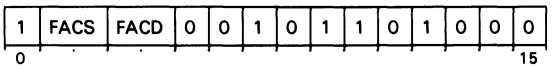


BITS	NAME	CONTENTS or FUNCTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator--while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator - while processing a floating point number, 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 FSCAL instruction, a significant bit was shifted out of the high order end of the mantissa; this bit is also set during a FIX instruction if the result does not fit into the destination location.
5	TE	Trap Enable - If this bit is 1, setting any of bits 1-4 will result in a floating point fault.
6	Z	Zero bit - The result of the last floating point operation was zero.
7	N	Negative bit--The result of the last floating point operation was less than zero.
8-11*	---	Reserved for future use.
12-15	FPMOD	Indicates computer series supporting the floating point instruction set. 0000 S/200, C/300, S/230, C/330 0001 S/130, S/250 standard FP 0010 M/600, C/350, S/250 optional FP 0011 Reserved for future use. 0100 Reserved for future use. 0101 8660 SP, 8661 SP 0110 C/150, S/250 standard EAU 0111- Reserved for 1111 future use.
16	---	Reserved for future use.
17-31	FPPC	Floating Point Program Counter - This is the logical address of the last floating point instruction executed. In the event of a floating point fault, this is the address of the floating point instruction that caused the fault.

**These bits are used as internal flags by the firmware; Preserve them when saving the state of the FPSR.*

Multiply Double (FPAC by FPAC)

FMD *facs,facd*



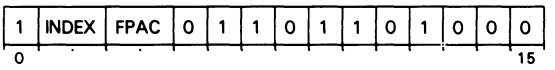
Multiplies the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FACD.

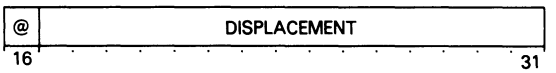
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 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 flag in the floating point status register 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 (FPAC by Memory)

FMMD *fpac,[@]displacement[,index]*





Multiplies the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

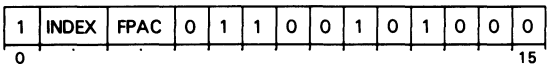
Computes the effective address, *E*, which addresses a 4-word (double precision) operand.

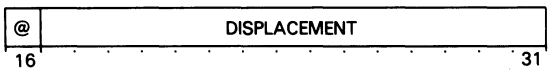
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 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 flag in the floating point status register 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 Single (FPAC by Memory)

FMMS *fpac,[@]displacement[,index]*





Multiplies the floating point number in FPAC by the floating point number in the source location and places the normalized result in FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

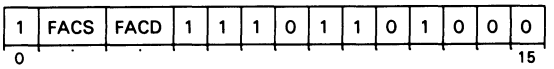
Computes the effective address *E* which addresses a 2-word single precision) operand.

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. 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 flag in the floating point status register 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.

Move Floating Point

FMOV *facs,facd*

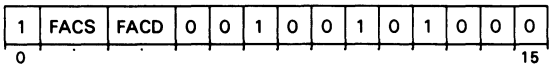


Moves the contents of one FPAC to another FPAC.

Places the contents of FACS in FACD, destroys the previous contents of FACD, and leaves the contents of FACS unchanged. If the mantissa in FACS is zero, the sign and exponent in FACD are also set to zero. The Z and N flags in the floating point status register are set to reflect the new contents of FACD.

Multiply Single (FPAC by FPAC)

FMS *facs,facd*



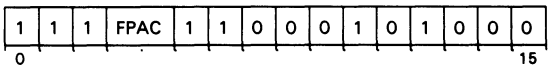
Multiplies the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FACD.

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. 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 flag in the floating point status register 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.

Negate

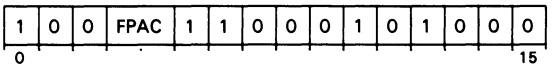
FNEG *fpac*



Inverts the sign bit of FPAC. Bits 1-63 of FPAC remain unchanged. Also sets the sign and exponent to zero if the mantissa in FPAC is zero. Updates the Z and N flags in the floating point status register to reflect the new contents of FPAC. If FPAC contains true zero, the sign bit remains unchanged.

Normalize

FNOM *fpac*

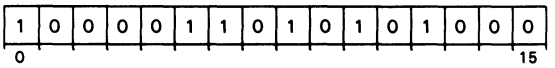


Normalizes the floating point numbers in FPAC. Sets a true zero in FPAC if all the bits of the mantissa are zero. Sets the UNF flag in the FPSR if an exponent underflow occurs. The number in FPAC is then correct, except that the exponent is 128 too large.

The Z and N flags in the floating point status register are set to reflect the new contents of FPAC.

No Skip

FNS

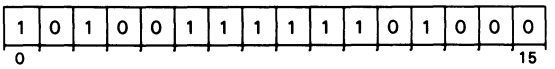


The next sequential word is executed.

Polynomial Evaluation Double

FPLYD

Optional Instruction (with FPP)



Evaluates a polynomial of a specified positive degree, and places the result in FPAC0. The inputs to the polynomial are as follows:

X	Original value in FPAC0.
N (degree)	Lower byte of word following instruction.
Coefficients	Following words.

Evaluates either normalized or unnormalized polynomials. (A normalized polynomial has coefficients adjusted so that the coefficient of the highest degree, A_n, is one.) If bit 0 of the word following the instruction is set to 1, the instruction evaluates a normalized polynomial. If bit 0 is 0, the instruction evaluates an unnormalized polynomial.

Polynomial: An unnormalized polynomial is of the form:

$$A_n X^n + A_{n-1} X^{n-1} + \ldots + A_1 X + A_0$$

A normalized polynomial is of the form:

$$X^n + A_{n-1} X^{n-1} + \ldots + A_1 X + A_0$$

Format: The coefficients of the polynomial must follow the instruction word and degree word. The format is:

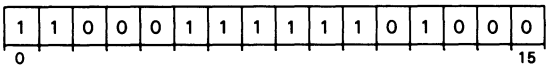
CODED WORD	MEANING
FPLYD	Instruction word
N	Nth degree, unnormalized
A_n	Coefficients
A_{n-1}	.
.	.
.	.
.	.
A_0	.
FPLYD	Instruction word
N + 100000_8	Nth degree, normalized
A_{n-1}	Coefficients
A_{n-2}	.
.	.
.	.
.	.
A_0	.

NOTE: *The first coefficient, A_n of a normalized polynomial is one. If a normalized polynomial has been specified, the algorithm does not expect you to supply A_n.*

Polynomial Evaluation Single

FPLYS

Optional Instruction (with FPP)



Evaluates a polynomial of a specified positive degree, and places the result in FPAC0. The inputs to the polynomial are as follows:

X	Original value in FPAC0.
N (degree)	Lower byte of word following instruction.
Coefficients	Following words.

Evaluates either normalized or unnormalized polynomials. (A normalized polynomial has coefficients adjusted so that the coefficient of the highest degree, A_n, is one.) If bit 0 of the word following the instruction is set to 1, the instruction evaluates a normalized polynomial. If bit 0 is 0, the instruction evaluates an unnormalized polynomial.

Polynomial: An unnormalized polynomial is of the form:

$$A_n X^n + A_{n-1} X^{n-1} + \ldots + A_1 X + A_0$$

A normalized polynomial is of the form:

$$X^n + A_{n-1} X^{n-1} + \ldots + A_1 X + A_0$$

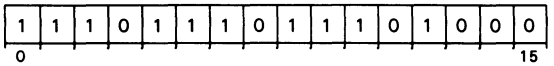
Format: The coefficients of the polynomial must follow the instruction word and degree word. The format is:

CODED WORD	MEANING
FPLYS	Instruction word
N	Nth degree, unnormalized
A_n	Coefficients
A_{n-1}	.
.	.
.	.
.	.
A_0	.
FPLYS	Instruction word
N + 100000_8	Nth degree, normalized
A_{n-1}	Coefficients
A_{n-2}	.
.	.
.	.
.	.
A_0	.

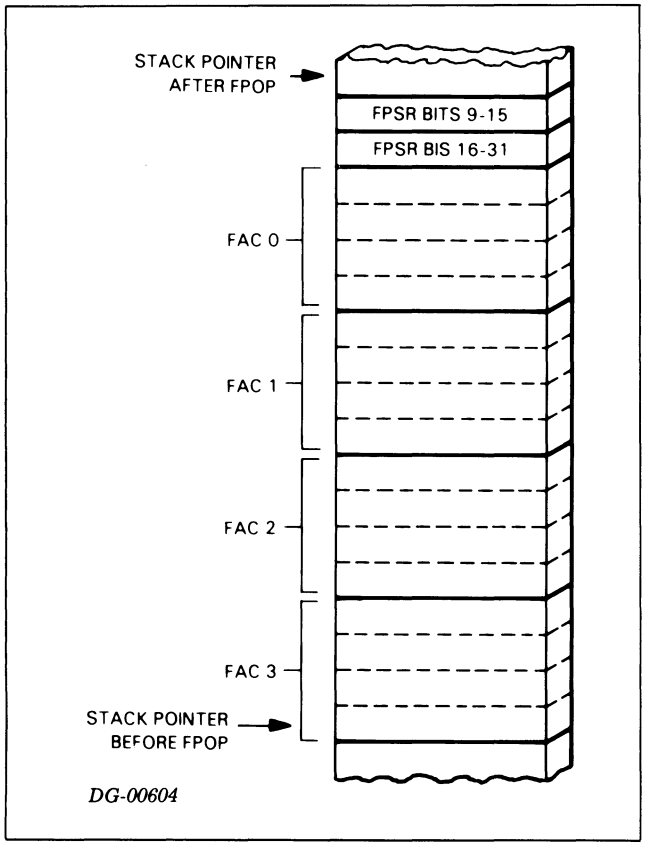
NOTE: *The first coefficient, A_n of a normalized polynomial is one. If a normalized polynomial has been specified, the algorithm does not expect you to supply A_n.*

Pop Floating Point State

FPOP



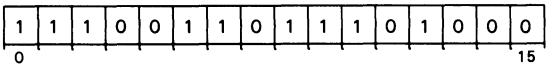
Pops an 18-word floating point return block off the user stack and alters the state of the floating point unit. The words popped and their destinations are as follows:



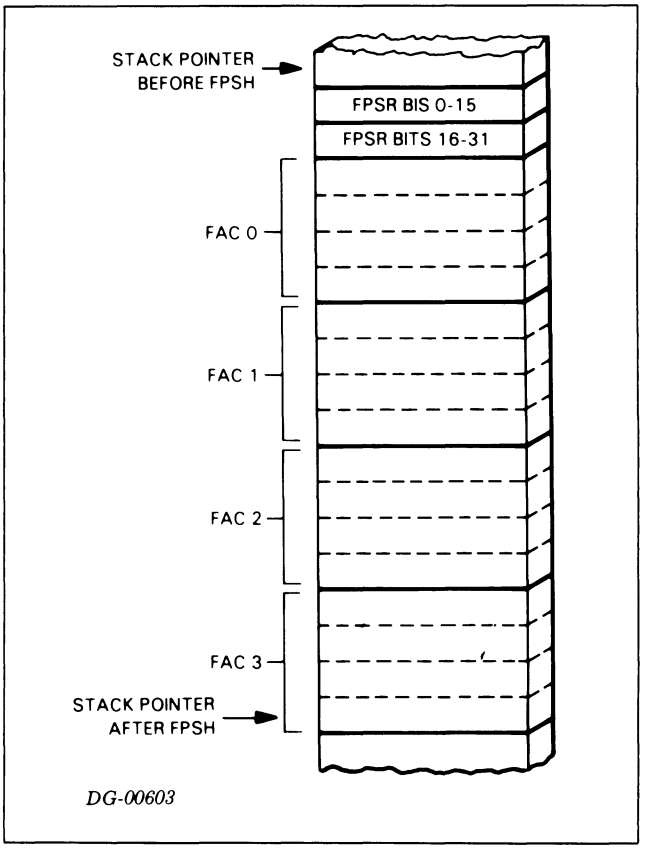
NOTE: *Because of the potentially long time required to perform some floating point instructions in relation to I/O interrupt requests, these instructions are interruptible. Because the FACD, stack pointer, and program counter are not updated until the completion of these instructions, any interrupt service routines that return control to the interrupted program via the program counter stored in location 0 will correctly restart these instructions.*

Push Floating Point State

FPSH

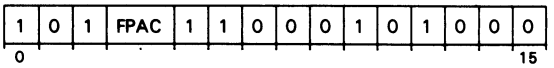


Pushes an 18-word floating point return block onto the user stack, leaving the contents of the floating point accumulators and the floating point status register unchanged. The format of the 18 words pushed is as follows:



Read High Word

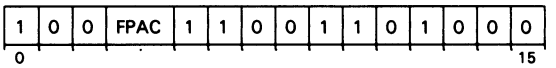
FRH *fpac*



Places the high-order 16 bits of FPAC in AC0, destroys the previous contents of AC0, and leaves unchanged the contents of FPAC and the Z and N flags in the floating point status register.

Scale

FSCAL *fpac*

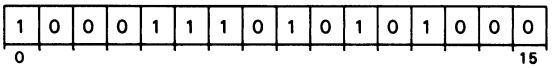


Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0. Leaves the contents of AC0 unchanged.

Treats bits 1-7 of AC0 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 AC0 bits 1-7. If the difference is zero, the instruction stops. 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; if bits are lost the instruction sets the MOF flag in the floating point status register. After the shift, the contents of bits 1-7 of AC0 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. The instruction sets the Z and N flags in the floating point status register to reflect the new contents of FPAC.

Skip Always

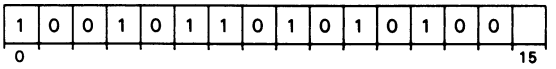
FSA



The next sequential word is skipped.

Skip On Zero

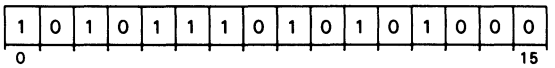
FSEQ



Skips the next sequential word if the Z flag of the floating point status register is 1.

Skip On Greater Than Or Equal To Zero

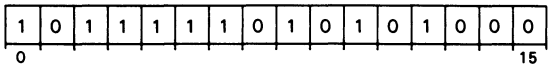
FSGE



Skips the next sequential word if the N flag of the floating point status register is 0.

Skip On Greater Than Zero

FSGT

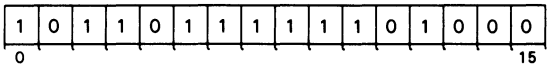


Skips the next sequential word if both the Z and N flags of the floating point status register are 0.

Sine Double

FSIND

Optional Instruction (with FPP)



Forms the sine of the number in FPAC0, places the result in FPAC0, and sets the Z and N flags of the floating point status register to reflect the new value in FPAC0. Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

The *Sine* and *Cosine* instructions can share the same data. The *Sine* instruction always skips the word immediately following the instruction word when searching for data. The *Cosine* instruction word can be placed in this location if desired.

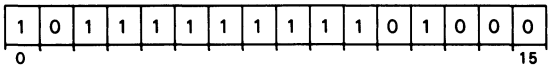
Format: Algorithm coefficients must follow the *Sine* instruction. The format is:

WORD	NAME	CODED VALUE (Hex)			
0	Instruction Word	FSIND			
1	Ignored	---			
2-5	4/PI	4114	5F30	6DC9	C883
6-9	A6	387C	F24A	053B	3668
10-13	A5	BA69	B262	61F8	B3A0
14-17	A4	3C3C	3E9F	5C1F	7D86
18-21	A3	BE15	5D3C	7DB7	837F
22-25	A2	3F40	F07C	206B	FE84
26-29	A1	C04E	F4F3	26F9	15EC
30-33	A0	40FF	FFFF	FFFF	FFCC
34-37	B6	3778	FBB4	E1B7	2DE0
38-41	B5	B978	C018	E66C	04DB
42-45	B4	3B54	1E0B	F28C	7BD1
46-49	B3	BD26	5A59	9C5A	A5E8
50-53	B2	3EA3	35E3	3BAC	37D9
54-57	B1	C014	ABBC	E625	BE3C
58-61	B0	40C9	0FDA	A221	6896

Sine Single

FSINS

Optional Instruction (with FPP)



Forms the sine of the number in FPAC0, places the result in FPAC0, and sets the Z and N flags of the floating point status register to reflect the new value in FPAC0. Places the contents of AC3 in the program counter and loads the value in location 41_8 (the frame pointer) into AC3.

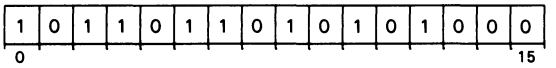
The *Sine* and *Cosine* instructions can share the same data base. The *Sine* instruction always skips the word immediately following the instruction word when searching for data. The *Cosine* instruction word can be placed in this location if desired.

Format: Algorithm coefficients must follow the *Sine* instruction. The format is:

WORD	NAME	CODED VALUE (Hex)	
0	Instruction Word	FSINS	
1	Ignored	---	
2-3	4/PI	4114	5F30
4-5	A3	BE14	E35E
6-7	A2	3F40	EBCA
8-9	A1	C04E	F4E3
10-11	A0	40FF	FFFF
12-13	B3	BD25	B25F
14-15	B2	3EA3	2F49
16-17	B1	C014	ABBC
18-19	B0	40C9	0FDB

Skip On Less Than Or Equal To Zero

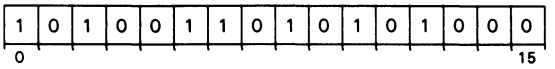
FSLE



Skips the next sequential instruction if either the Z flag or the N flag of the floating point status register is 1.

Skip On Less Than Zero

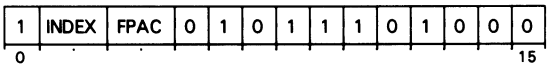
FSLT

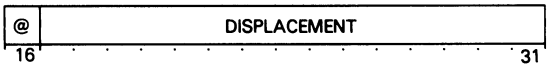


Skips the next sequential word if the N flag of the floating point status register is 1.

Subtract Double (Memory from FPAC)

FSMD *fpac,[@]displacement[,index]*





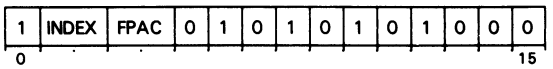
Subtracts the floating point number in the source location from the floating point number in FPAC and places the normalized result in the FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

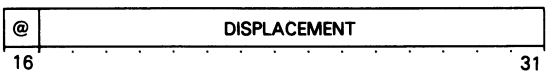
The instruction computes the effective address, *E*, which addresses a 4-word (double precision) operand.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition. (*See* FAMD.)

Subtract Single (Memory from FPAC)

FSMS *fpac,[@]displacement[,index]*





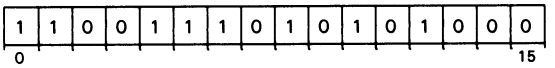
Subtracts the floating point number in the source location from the floating point number in FPAC and places the normalized result in the FPAC. Destroys the previous contents of FPAC, leaves the contents of the source location unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

The instruction computes the effective address, *E*, which addresses a 2-word (single precision) operand.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition. (*See* FAMS.)

Skip On No Zero Divide

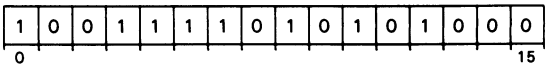
FSND



Skips the next sequential word if the divide by zero (DVZ) flag of the floating point status register is 0.

Skip On Non-Zero

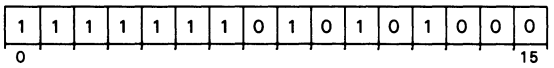
FSNE



Skips the next sequential word if the Z flag of the floating point status register is 0.

Skip On No Error

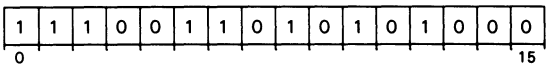
FSNER



Skips the next sequential word if bits 1-4 of the floating point status register are all 0.

Skip On No Overflow

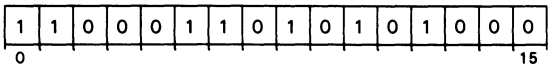
FSNO



Skips the next sequential word if the overflow (OVF) flag of the floating point status register is 0.

Skip On No Mantissa Overflow

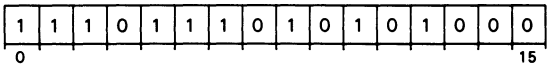
FSNM



Skips the next sequential word if the mantissa overflow (MOF) flag of the floating point status register is 0.

Skip On No Overflow and No Zero Divide

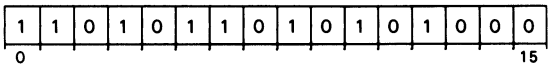
FSNOD



Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the floating point status register are 0.

Skip On No Underflow

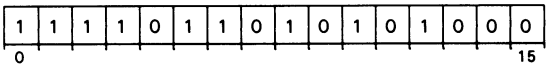
FSNU



Skips the next sequential word if the underflow (UNF) flag of the floating point status register is 0.

Skip On No Underflow And No Overflow

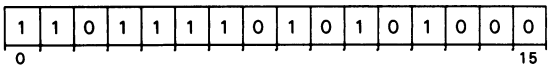
FSNUO



Skips the next sequential word if both the underflow (UNF) flag and overflow (OVF) flag of the floating point status register are 0.

Skip On No Underflow And No Zero Divide

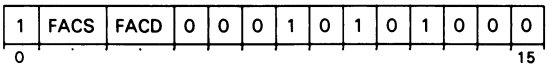
FSNUD



Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the floating point status register are 0.

Subtract Single (FPAC from FPAC)

FSS *facs,facd*



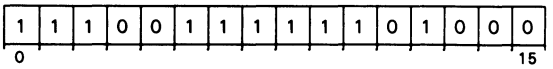
Subtracts the floating point number in FACS from the floating point number in FACD and places the normalized result in the FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents of FACD.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition.

Square Root Double

FSQRD

Optional Instruction (with FPP)



Forms the square root of the number in FPAC0 and puts the result into FPAC0. Sets the Z and N flags of the floating point status register to reflect the new value in FPAC0.

Normal return: Places the contents of AC3 in the program counter and loads the contents of location 41_8 (the frame pointer) into AC3.

Error return: Occurs if the original value in FPAC0 is negative (the square root function is invalid for these values). Loads the address of the word following the *Square Root* instruction into the program counter.

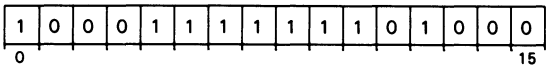
Format: Use the following format:

NAME	CODED VALUE
FSQRD ERRTN	Instruction word Control goes to address ERRTN if FPAC0<0

Square Root Single

FSQRS

Optional Instruction (with FPP)



Forms the square root of the number in FPAC0 and puts the result into FPAC0. Sets the Z and N flags of the floating point status register to reflect the new value in FPAC0.

Normal return: Places the contents of AC3 in the program counter and loads the contents of location 41_8 (the frame pointer) into AC3.

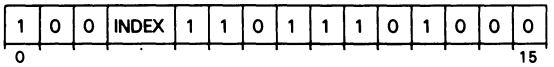
Error return: Occurs if the original value in FPAC0 is negative (the square root function is invalid for these values). Loads the address of the word following the *Square Root* instruction into the program counter.

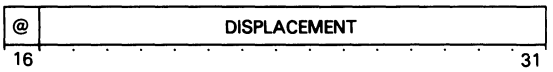
Format: Use the following format:

NAME	CODED VALUE
FSQRS ERRTN	Instruction word Control goes to address ERRTN if FPAC0<0

Store Floating Point Status

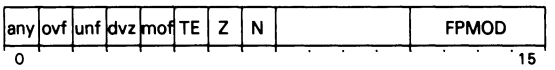
FSST *[@]displacement[,index]*

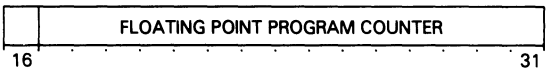




Moves the contents of the FPSR to two specified memory locations.

Computes the effective address, *E*, and places the 32-bit contents of the FPSR in the two consecutive memory locations addressed by *E* and *E* + *1*. Leaves the contents of the FPSR unchanged.



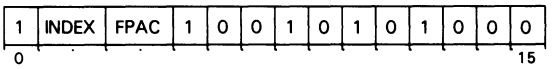


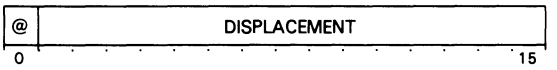
BITS	NAME	CONTENTS or FUNCTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator--while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator - while processing a floating point number, 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 FSCAL instruction, a significant bit was shifted out of the high order end of the mantissa; this bit is also set during a FIX instruction if the result does not fit into the destination location.
5	TE	Trap Enable - If this bit is 1, setting any of bits 1-4 will result in a floating point fault.
6	Z	Zero bit - The result of the last floating point operation was zero.
7	N	Negative bit--The result of the last floating point operation was less than zero.
8-11*	---	Reserved for future use.
12-15	FPMOD	Indicates computer series supporting the floating point instruction set. 0000 S/200, C/300, S/230, C/330 0001 S/130, S/250 standard FP 0010 M/600, C/350, S/250 optional FP 0011 Reserved for future use. 0100 Reserved for future use. 0101 8660 SP, 8661 SP 0110 C/150, S/250 standard EAU 0111- Reserved for 1111 future use.
16	---	Reserved for future use.
17-31	FPPC	Floating Point Program Counter - This is the logical address of the last floating point instruction executed. In the event of a floating point fault, this is the address of the floating point instruction that caused the fault.

**These bits are used as internal flags by the firmware; Preserve them when saving the state of the FPSR.*

Store Floating Point Single

FSTS *fpac,[@]displacement[,index]*



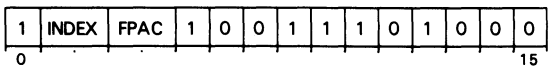


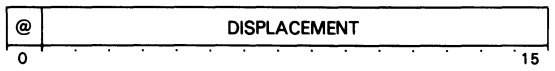
Stores the contents of a specified FPAC into a memory location.

Computes the effective address *E* and places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location and leaves unchanged the contents of FPAC and the condition codes in the FPSR. For single precision, only the high-order 32 bits of FPAC are stored.

Store Floating Point Double

FSTD *fpac,[@]displacement[,index]*



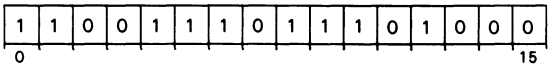


Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*, and places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location and leaves unchanged the contents of FPAC and the condition codes in the FPSR.

Trap Disable

FTD

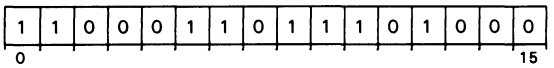


Sets the trap enable bit of the FPSR to 0.

NOTE: *The I/O RESET instruction will set this bit to 0.*

Trap Enable

FTE

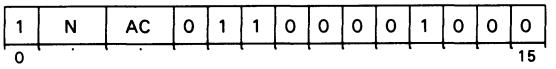


Sets the trap enable bit of the FPSR to 1.

NOTE: *When a floating point fault occurs and the trap enable bit is 1, the trap enable bit is set to 0 before control is transferred to the floating point error handler. The trap enable bit should be set to 1 before normal processing is resumed.*

Hex Shift Left

HXL *n,ac*

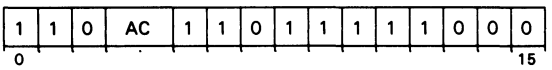


Shifts the contents of AC left a number of hex digits depending upon the immediate field N. The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of AC are set to 0.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Halve

HLV *ac*



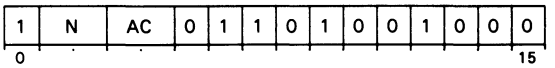
Divides the contents of an accumulator by 2 and rounds the result toward zero.

The signed, 16-bit two's complement number contained in the specified AC is divided by 2 and rounded toward 0. The result is placed in the specified AC.

If the number is positive, division is accomplished by shifting the number right one bit. If the number is negative, division is accomplished by negating the number, shifting it right one bit, and negating it again.

Hex Shift Right

HXR *n,ac*

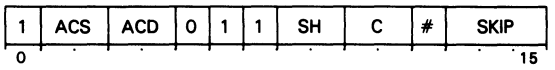


Shifts the contents of AC right a number of hex digits depending upon the immediate field, N. The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of AC are set to 0.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Increment

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



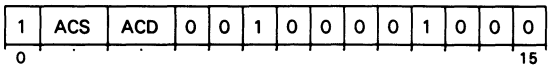
Increments the contents of an accumulator.

Initializes carry 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 carry. 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 carry.*

Inclusive OR

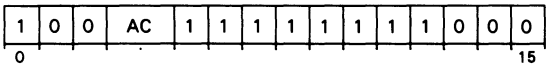
IOR *acs,acd*

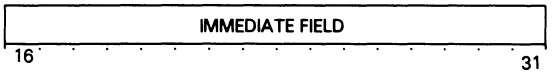


Forms the logical inclusive OR of the contents of ACS and the contents of ACD and places the result in ACD. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the instruction sets the result bit to 0. The contents of ACS remain unchanged.

Inclusive OR Immediate

IORI *i,ac*

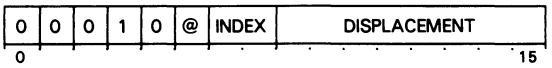




Forms the logical inclusive OR of the contents of the immediate field and the contents of the specified AC and places the result in the specified AC.

Increment And Skip If Zero

ISZ *[@]displacement[,index]*

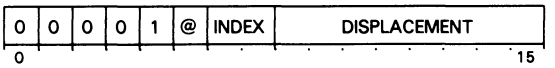


Increments the addressed word, then skips if the incremented value is zero.

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.

Jump To Subroutine

JSR *[@]displacement[,index]*



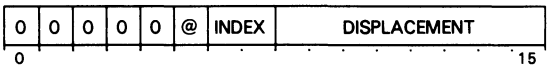
Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

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.*

Jump

JMP

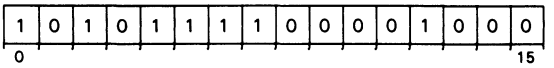


Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Load Control Store Formatted

LCSF

Optional Instruction (with WCS)



Forms a series of microinstructions from the contents of sequential memory locations, and places them in the control store.

AC1 contains a count of the number of microinstructions to be loaded.

Bits 1-15 of AC2 contain the address of the first source location in main memory.

Bits 4-15 of AC3 contain the address of the first destination location, or control store address.

If bit 0 of either AC2 or AC3 is 1, then the address contained in bits 1-15 is an indirect address in main memory. The instruction resolves the indirection chain and places the effective address in the appropriate accumulator before the data movement occurs.

The resolved address in AC3 addresses a location in the WCS memory. The WCS starts at microaddress 4000_8 and extends to 5777_8. Since microinstructions require 57 bits, four 16-bit words are required for each

microcode word. The contents of these four 16-bit words are entered in the microcode word as follows:

The complements of bits 0 through 13 of the first word in main memory are loaded into bits 0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48 and 52 of the first microcode word.

The complements of bits 0 through 13 of the second word in main memory are loaded into bits 1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49 and 53 of the first microcode word.

The complements of bits 0 through 13 of the third word in main memory are loaded into bits 2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50 and 54 of the first microcode word.

The complements of bits 0 through 14 of the fourth word in main memory are loaded into bits 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55 and 56 of the first microcode word.

The next four words in main memory similarly are loaded into the second microcode word, and so on.

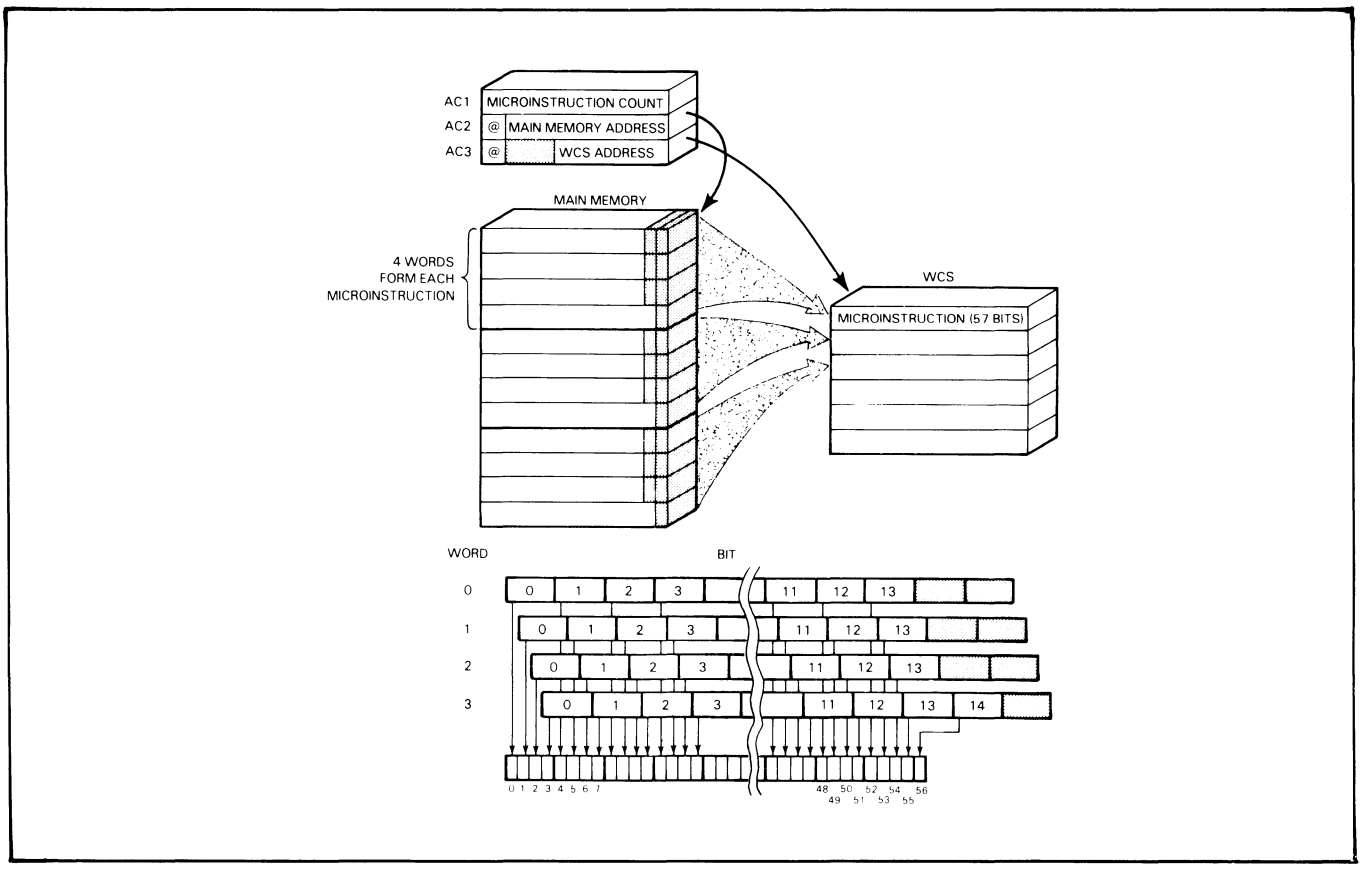
For each microcode word move, the count in AC1 is decremented by one, the source address in AC2 is incremented by four, and the destination address in AC3 is incremented by one. Upon completion of the instruction, AC1 contains zero, and AC2 and AC3 point to the word following the last word in their respective fields. The contents of AC0 remain unchanged.

Words are moved in consecutive ascending order according to their addresses. The next source address after 77777_8 is 0. If at any the time the destination address is outside the range of 4000_8 to 5777_8, the instruction terminates.

NOTES: *Due to the potentially long time that may be required to perform this instruction in relation to I/O requests, this instruction is interruptable. If an LCSF instruction is interrupted, program counter is decremented by one before it is placed in location 0 so that it points to the LCSF instruction. Because the addresses and the word count are updated after every word stored, an interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the LCSF instruction.*

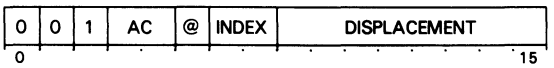
The LCSF instruction can be disabled for any user by enabling I/O protection for that user with the MAP. An attempt by this user to use the LCSF instruction will cause a protection fault.

When updating the source and destination addresses, the LCSF instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the LCSF instruction will not try to resolve an indirect address in either AC2 or AC3.



Load Accumulator

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

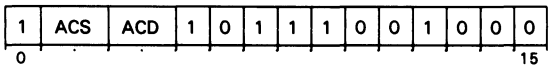


Copies a word from memory to an accumulator.

Places the word addressed by the effective address, *E*, in the specified accumulator. The previous contents of the location addressed by *E* remain unchanged.

Load Byte

LDB *acs,acd*

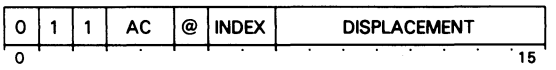


Moves a copy of the contents of a memory byte (as addressed by a byte pointer in one accumulator) into the second accumulator.

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.

Load Effective Address

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



Computes the effective address, *E*, and places it in bits 1-15 of the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the AC are lost.

If you reference an auto-incrementing or auto-decrementing location during the effective address calculation, the instruction increments or decrements as appropriate the contents of the auto-incrementing or -decrementing location.

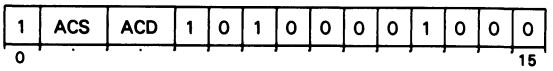
```
LEF    0,TABLE      ; The logical address of
                    ; TABLE is placed in AC0.
LEF    1,-55,3      ; Subtracts 000055 (octal)
                    ; from the unsigned integer
                    ; in AC3 and the result is
                    ; placed in AC1.
LEF    0,.+0        ; Places the address of this
                    ; Load effective address
                    ; instruction in AC0.
```

NOTE: *The* **LEF** *instruction can only be used in a mapped system, while in the user mode. With the* **LEF** *mode bit set to 1, all I/O and* **LEF** *instructions will be interpreted as* **LEF** *instructions. With the* **LEF** *mode bit set to 0, all I/O and* **LEF** *instructions will be interpreted as I/O instructions.*

Be sure that I/O protection is enabled or the **LEF** *mode bit is set to 1 before using the* **LEF** *instruction. If you issue a* **LEF** *instruction in the I/O mode, with protection disabled, the instruction will be interpreted and executed as an I/O instruction, with possibly undesirable results.*

Locate Lead Bit

LOB *acs,acd*

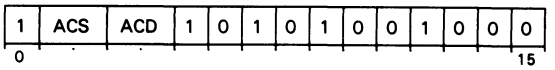


Adds a number equal to the number of high-order zeroes in the contents of ACS to the signed, 16-bit, two's complement number contained in ACD. The contents of ACS and the state of carry remain unchanged.

NOTE: *If ACS and ACD are specified as the same accumulator, the instruction functions as described above, except that since ACS and ACD are the same accumulator, the contents of ACS will be changed.*

Locate and Reset Lead Bit

LRB *acs,acd*



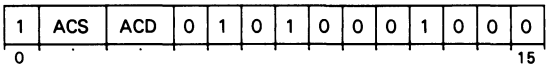
Performs a *Locate lead bit* instruction, and sets the lead bit to 0.

Adds a number equal to the number of high-order zeroes in the contents of ACS to the signed, 16-bit, two's complement number contained in ACD. Sets the leading 1 in ACS to 0. The state of carry remains unchanged.

NOTE: *If ACS and ACD are specified to be the same accumulator, then the instruction sets the leading 1 in that accumulator to 0, and no count is taken.*

Logical Shift

LSH *acs,acd*



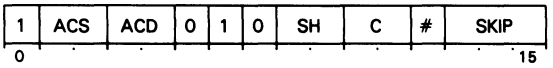
Shifts the contents of ACD either left or right depending on the number contained in bits 8-15 of ACS. The signed, 8-bit two's complement number contained in bits 8-15 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 8-15 of ACS is positive, shifting is to the left; if the number in bits 8-15 of ACS is negative, shifting is to the right. If the number in bits 8-15 of ACS is zero, no shifting is performed. Bits 0-7 of ACS are ignored.

The number of bits shifted is equal to the magnitude of the number in bits 8-15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. The carry bit and the contents of ACS remain unchanged.

NOTE: *If the magnitude of the number in bits 8-15 of ACS is greater than 15, all bits of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Move

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

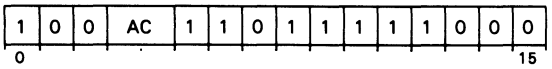


Moves the contents of an accumulator through the Arithmetic Logic Unit (ALU).

Initializes carry 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.

Modify Stack Pointer

MSP *ac*



Changes the value of the stack pointer and tests for potential overflow.

Adds the signed two's-complement number in the specified accumulator to the value of the stack pointer and places the result in location 40. The instruction then checks for overflow by comparing the result in location 40with the value of the stack limit. If the result in location 40 is less than the stack limit, then the instruction ends.

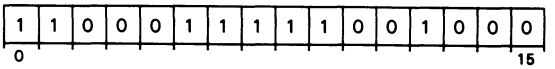
If the result is greater than the stack limit, the instruction changes the value of location 40 back to its original contents before the add. The instruction pushes a return block of the format shown below:

The program counter in the return block contains the address of the *Modify Stack Pointer* instruction.

After pushing the return block, the program counter contains the address of the stack fault routine. The stack pointer is updated with the value used to push the return block, and control transfers to the stack fault routine.

Unsigned Multiply

MUL

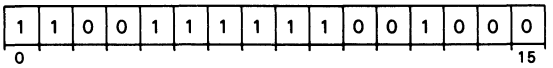


Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

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

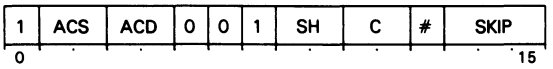


Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

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.

Negate

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



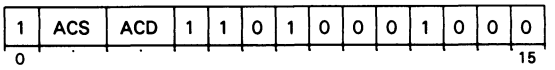
Forms the two's complement of the contents of an accumulator.

Initializes carry 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 carry. 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 carry.*

Pop Multiple Accumulators

POP *acs,acd*



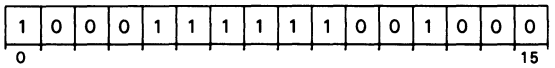
Pops 1 to 4 words off the stack and places them in the indicated accumulators.

The set of accumulators from ACS through ACD is filled with words popped from the stack. The accumulators are filled in descending order, starting with the AC specified by ACS and continuing down through the AC specified by ACD, wrapping around if necessary, with AC3 following AC0. If ACS is equal to ACD, only one word is popped and it is placed in ACS.

The stack pointer is decremented by the number of accumulators popped and the frame pointer is unchanged. A check for underflow is made only after the entire pop operation is done.

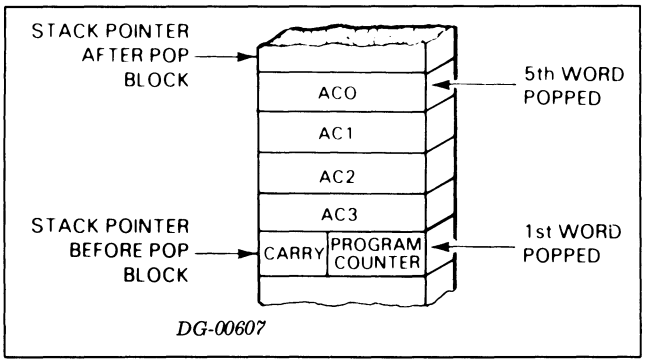
Pop Block

POPB



Returns control from a *System Call* routine or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction.

Five words are popped off the stack and placed in predetermined locations. The words popped and their destinations are as follows:

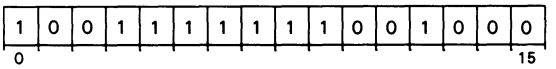


Sequential operation is continued with the word addressed by the updated value of the program counter.

> NOTE: *If the I/O handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction, do not use the* Pop Block *instruction. Use the* Restore *instruction instead.*

Pop PC And Jump

POPJ

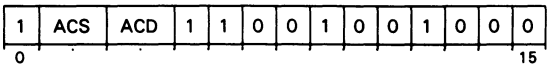


Pops the top word off the stack and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The stack pointer is decremented by one and the frame pointer is unchanged. A check for underflow occurs after the pop operation.

Push Multiple Accumulators

PSH *acs,acd*



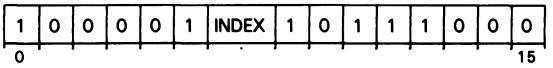
Pushes the contents of 1 to 4 accumulators onto the stack.

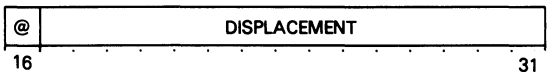
The set of accumulators from ACS through ACD is pushed onto the stack. The accumulators are pushed in ascending order, starting with the AC specified by ACS and continuing up through the AC specified by ACD, wrapping around if necessary, with AC0 following AC3. The contents of the accumulators remain unchanged. If ACS equals ACD, only ACS is pushed.

The stack pointer is incremented by the number of accumulators pushed and the frame pointer is unchanged. A check for overflow is made only after the entire push operation finishes.

Push Jump

PSHJ *[@]displacement[,index]*

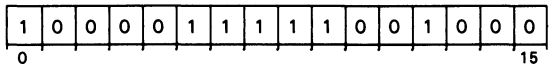




Pushes the address of the next sequential instruction onto the stack, computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Push Return Address

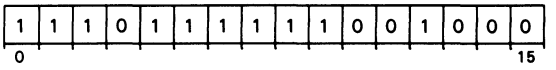
PSHR



Pushes the address of this instruction *plus 2* onto the stack.

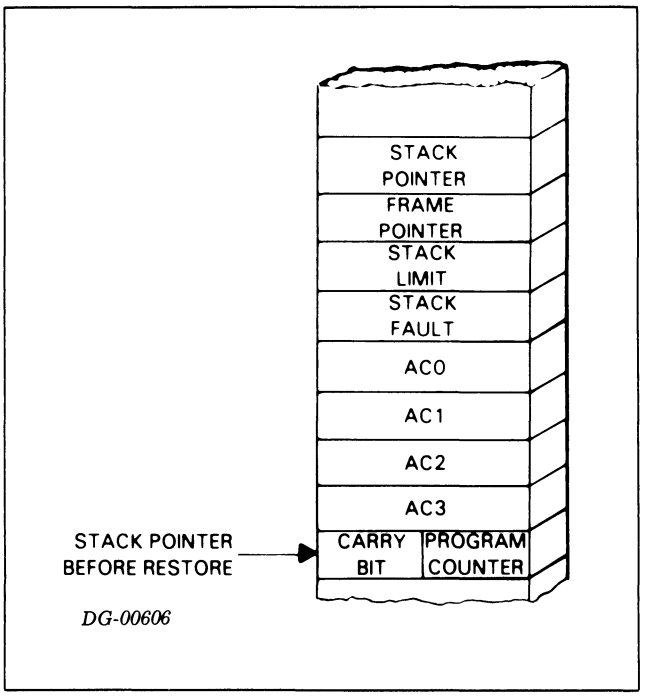
Restore

RSTR



Returns control from certain types of I/O interrupts.

Pops nine words off the stack and places them in predetermined locations. The words popped and their destinations are as follows:



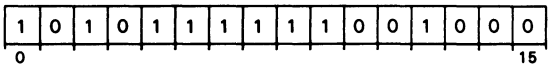
Sequential operation continues with the word addressed by the updated value of the program counter.

NOTE: *Use the* Restore *instruction to return control to the program only if the I/O interrupt handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction.*

The Restore *instruction does not check for stack underflow.*

Return

RTN



Returns control from subroutines that issue a *Save* instruction at their entry points.

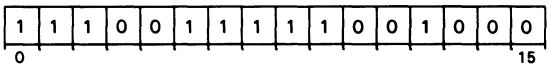
The *Save* instruction loads the current value of the stack pointer into the frame pointer. The *Return* instructions uses this value of the frame pointer to pop a standard return block off of the stack. The format of the return block is:

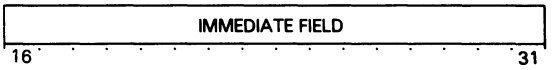
WORD POPPED	DESTINATION
1	Bit 0 is loaded into carry Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

After popping the return block, the *Return* instruction loads the decremented value of the frame pointer into the stack pointer and the popped value of AC3 into the frame pointer.

Save

SAVE *i*





Saves the information required by the *Return* instruction.

Saves the current value of the stack pointer in a temporary location. Adds five plus the unsigned, 16-bit integer contained in the immediate field to the current value of the stack pointer and loads the result into location 40. Compares this new value of the stack pointer to the stack limit to check for overflow. If no overflow condition exists, then the instruction places the current value of the frame pointer in AC3. Fetches the contents of the temporary location and loads them into the frame pointer. The instruction uses the value in the frame pointer to push a five-word return block. The formats and contents of the five-word return block 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

After pushing the return block, the instruction places the value of the frame pointer (which now contains the old value of the stack pointer plus five) in AC3.

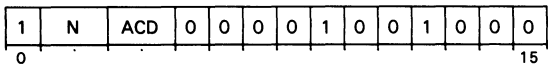
If an overflow condition exists, the *Save* instruction transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save* instruction.

The *Save* instruction allocates a portion of the stack for use by the procedure which executed the *Save*. The value of the *frame size*, contained in the immediate field, determines the number of words in this stack area. This portion of the stack will not normally be accessed by push and pop operations, but will be used by the procedure for temporary storage of variables, counters, etc. The frame pointer acts as the reference point for this storage area.

Use the *Save* instruction with the *Jump to Subroutine* instruction. The *Jump to Subroutine* instruction places the return value of the program counter in AC3. *Save* then pushes the return value (contents of AC3) into

Subtract Immediate

SBI *n,ac*

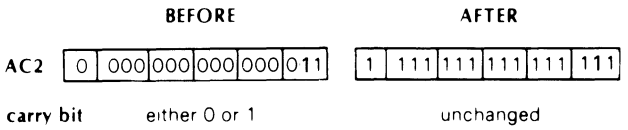


Subtracts an unsigned integer in the range 1-4 from the contents of an accumulator.

The contents of the immediate field *N*, plus 1 are subtracted from the unsigned 16-bit number contained in the specified AC and the result is placed in ACD. Carry remains unchanged.

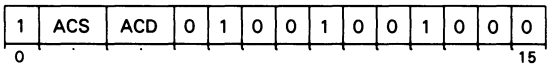
> NOTE: *The assembler takes the coded value of n and subtracts one from it before placing it in the immediate field. Therefore code the exact value you wish to subtract.*

Example - Assume that AC2 contains 000003_8. After the instruction **SBI 4,2** is executed, AC2 contains 177777_8 and carry remains unchanged.



Skip If ACS Greater Than Or Equal to ACD

SGE *acs,acd*



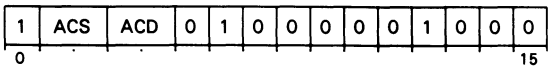
Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

The signed two's complement numbers in ACS and ACD are algebraically compared. If the number in ACS is greater than or equal to the number in ACD, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

> NOTE: *The* Skip If ACS Greater Than ACD *and* Skip If ACS Greater Than Or Equal To ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the Subtract and Add complement instructions.*

Skip If ACS Greater Than ACD

SGT *acs,acd*

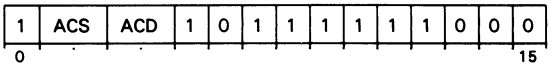


Compares two signed integers in two accumulators and skips if the first is greater than the second.

The signed, two's complement numbers in ACS and ACD are algebraically compared. If the number in ACS is greater than the number in ACD, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

Skip On Non-Zero Bit

SNB *acs,acd*



The two accumulators form a bit pointer. If the addressed bit is 1, the next sequential word is skipped.

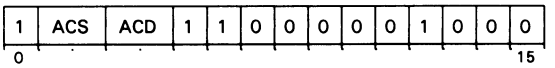
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 1, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

NOTE: *The bit pointer formed by the two accumulators cannot make indirect memory references.*

Store Byte

STB *acs,acd*

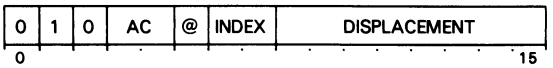


Moves the right byte of ACD to a byte in memory. ACS contains the byte pointer.

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.

Store Accumulator

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

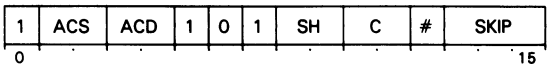


Stores the contents of an accumulator into a memory location.

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]*



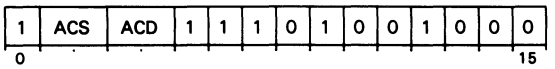
Performs unsigned integer subtraction and complements carry if appropriate.

Initializes carry 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 carry. 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 carry.*

System Call

SYC *acs,acd*



Pushes a return block and transfers control to the *system call handler.*

If a user map is enabled, the instruction disables it and pushes a return block onto the stack. The program counter in the return block points to the instruction immediately following the *System call* instruction. After pushing the return block, the instruction executes a *jump indirect* to location 2, which contains the address of the *system call handler.*

If this instruction disables a user map, then I/O interrupts cannot occur between the time the *System call* instruction is executed and the time the first instruction of the system call handler is executed.

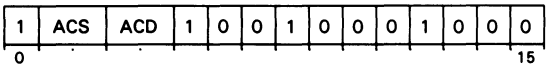
NOTE: *If both accumulators are specified as AC0, the instruction does not push a return block onto the stack. The contents of AC0 remain unchanged.*

The assembler recognizes the mnemonic **SCL** *as equivalent to* **SYC 1,1.**

The assembler recognizes the mnemonic **SVC** *as equivalent to* **SYC 0,0.**

Skip On Zero Bit

SZB *acs,acd*



The two accumulators form a bit pointer. If the addressed bit is zero, the next sequential word is skipped.

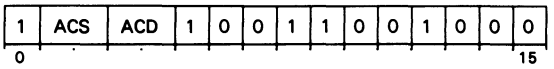
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 0, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

NOTE: *The bit pointer contained in ACS and ACD cannot make indirect memory references.*

Skip On Zero Bit And Set To One

SZBO *acs,acd*



The two accumulators form a bit pointer. The instruction sets the addressed bit to 1. If the addressed bit was 0 before being set to 1, the instruction skips the next sequential word. The contents of ACS and ACD remain unchanged.

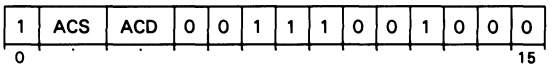
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

NOTE: *The bit pointer contained in ACS and ACD must not make indirect memory references*

This instruction facilitates the use of bit maps for such purposes as allocation of facilities (memory blocks, I/O devices, etc.) to several processes, or tasks, that may interrupt one another, or in a multiprocessor environment. The bit is tested and set to 1 in one memory cycle.

Exchange Accumulators

XCH *acs,acd*

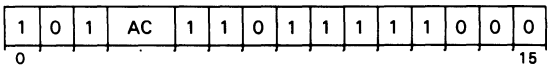


Exchanges the contents of two accumulators.

Places the original contents of ACS in ACD and the original contents of ACD in ACS.

Execute

XCT *ac*



Executes the instruction contained in AC as if it were in main memory in the location occupied by the *Execute* instruction. If the instruction in AC is an *Execute* instruction which executes the instruction in AC, the processor is placed in a one-instruction loop. The Stop switch on the console will not stop the processor, but the Reset switch will.

Because of the possibility of AC containing an *Execute* instruction, this instruction is interruptible. An I/O interrupt can occur immediately prior to each time the instruction in AC is executed. If an I/O interrupt does occur, the program counter in the return block pushed on the system stack points to the *Execute* instruction in main memory. This capability to execute an *Execute* instruction gives you a *wait for I/O interrupt* instruction.

NOTE: *If the specified accumulator contains the first word of a two-word instruction, the word following the* **XCT** *instruction is used as the second word. Normal sequential operation then continues from the second word after the* **XCT** *instruction.*

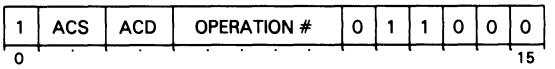
Do not use the **XCT** *instruction to execute an instruction that requires all four accumulators, such as* **CMV, CMT, CMP, CTR,** *or* **BAM.**

The results of **XCT** *are undefined if the specified accumulator contains an instruction that modifies that same accumulator. For example:*

```
        LDA   0,TOT
        XCT   0              ;UNDEFINED
        JMP   ON
TOT:    ADD   1,0
```

Extended Operation

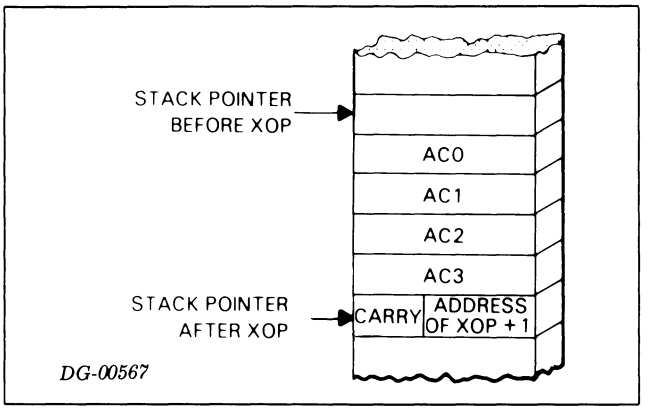
XOP *acs,acd,entry number*



Pushes a return block onto the stack. Places the address in the stack of ACS into AC2; places the address in the stack of ACD into AC3. Memory location 44_8 must contain the XOP origin address, the starting address of a 32_{10} word table of addresses. These addresses are the starting location of the various XOP operations.

Adds the operation number in the XOP instruction to the XOP origin address to produce the address of a word in the XOP table. The instruction fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the XOP origin address remain unchanged.

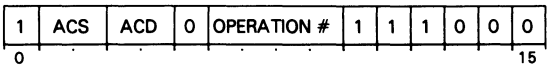
The format of the return block pushed by the XOP instruction is as follows:



This return block is configured so that the XOP procedure can return control to the calling program via the POP BLOCK instruction.

Enter WCS

XOP1 *acs,acd,entry number*

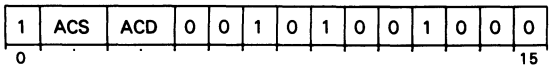


Executes the microprogram in WCS whose entry number corresponds to bits 6-9 in the *Enter WCS* instruction. Permissible entry numbers are 0-15, corresponding to the first 16 locations in WCS. The executed microprogram controls the use of accumulators, whether or not they are changed, and the location of the next instruction.

If the WCS feature is not installed, the *Enter WCS* instruction operates exactly like the *Extended Operation* instruction except that it adds 32_{10} to the entry number before it adds the entry number to the XOP origin address.

Exclusive OR

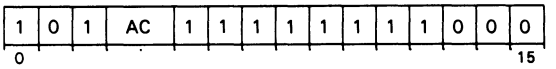
XOR *acs,acd*

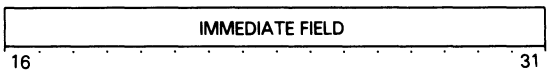


Forms the logical exclusive OR of the contents of ACS and the contents of ACD and places the result in ACD. Sets a bit position in the result to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the instruction sets result bit to 0. The contents of ACS remain unchanged.

Exclusive OR Immediate

XORI *i,ac*





Forms the logical exclusive OR of the contents of the immediate field and the contents of the specified AC and places the result in the specified AC.

Chapter V
ECLIPSE S/250 I/O INSTRUCTIONS

This chapter lists the ECLIPSE S/250 I/O instructions, plus those intended for a specific device such as the MAP, the BMC, and the CPU. We have arranged these instructions in alphabetical order according to mnemonics as recognized by the assembler.

For each instruction we include:

- the mnemonic recognized by the assembler
- the bit format required
- the format of any arguments involved
- the functional description of each instruction

In general, these I/O instructions can be executed only with *Lef* mode and I/O protection disabled. See the Memory Allocation and Protection section in Chapter II for a discussion of *Lef* mode and I/O protection.

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

GENERAL I/O INSTRUCTIONS

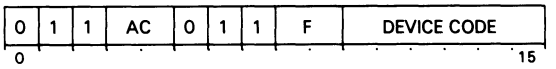
You can use the following general I/O instructions with any I/O device, using the appropriate device code.

Device Flag Commands

f=S	Issues a Start pulse to the specified device.
f=C	Issues a Clear pulse to the specified device.
f=P	Issues an I/O pulse to the specified device.
IORST	No effect.

Data in B

DIB *[f] ac,device*



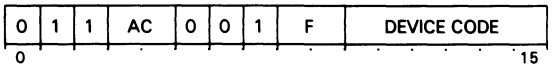
Transfers data from the B buffer of an I/O device to an accumulator.

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 In A

DIA *[f] ac,device*



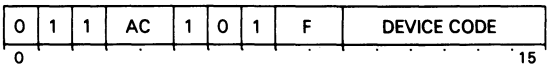
Transfers data from the A buffer of an I/O device to an accumulator.

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*



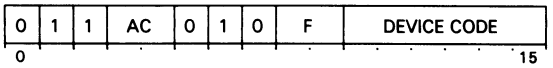
Transfers data from the C buffer of an I/O device to an accumulator.

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 Out A

DOA *[f] ac,device*



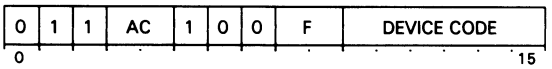
Transfers data from an accumulator to the A buffer of an I/O device.

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*



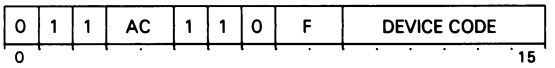
Transfers data from an accumulator to the B buffer of an I/O device.

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*



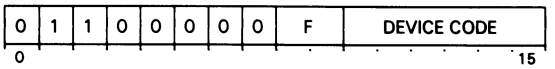
Transfers data from an accumulator to the C buffer of an I/O device.

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.

No I/O Transfer

NIO *[f] ac,device*

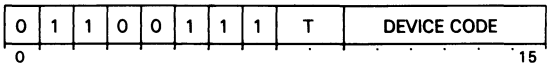


Used when a Busy or Done flag must be changed with no other operation taking place.

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

I/O Skip

SKP *[t] device*



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

BURST MULTIPLEXOR CHANNEL

Device Code - 5_8 (Primary)

Priority Mask Bit - None

Device Flag Commands

f=S — Sets the Busy flag to 1 and initiates a BMC map load or dump sequence.

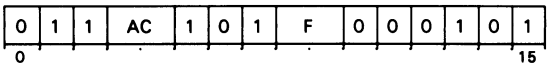
f=C — Sets the status register (except bit 1) to 0.

f=P — No effect.

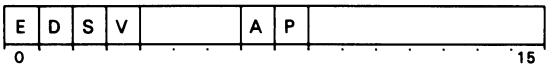
IORST — Sets the status register (except bit 1) to 0; turns off mapping.

Read Status

DIC *[f]* *ac*,**BMC**



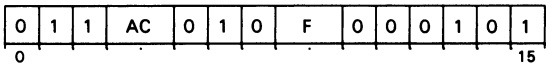
Loads the burst multiplexor status flags into the specified accumulator. The previous contents of the accumulator are lost. The format of the accumulator is shown below.



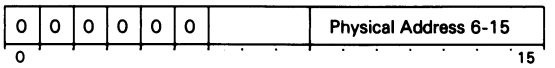
BITS	NAME	CONTENTS or FUNCTION
0	E	When 1, the channel has detected a validity protect error, an address parity error, or a data parity error.
1	D	When 1, the direction for a map data transfer is from the register(s) to memory (dump).
2	S	When 1, the channel is in two step diagnostic mode.
3	V	When 1, the channel has detected a validity protect error.
4-6	---	Reserved for future use.
7	A	When 1, the channel has detected an address parity error.
8	P	When 1, the channel has detected a data parity error.
9-15	---	Reserved for future use.

Specify Low-Order Address

DOA *[f]* *ac*,**BMC**



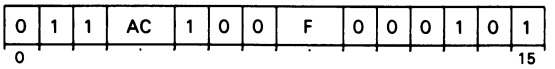
The contents of the specified accumulator specify the low-order 10 bits of the 20-bit physical memory address of the first word to be transferred to or from the map. The contents of the accumulator are unchanged. The format of the accumulator is shown below.



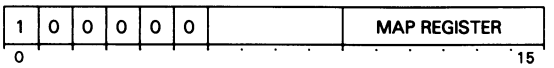
BITS	NAME	CONTENTS or FUNCTION
0-5	---	Must be 0.
6-15	LO ADDR	Specify the least significant bits of the physical address for the start of a map data transfer.

Specify Initial Map Register

DOB *[f]* *ac*,**BMC**



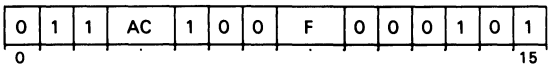
The contents of the specified accumulator select the first map register to be loaded or dumped in the next map data transfer. The contents of the accumulator are unchanged. The format of the accumulator is shown below.



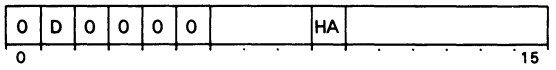
BITS	NAME	CONTENTS or FUNCTION
0	---	Must be 1.
1-5	---	Must be 0.
6-15	MAP REGISTER	Specify a map register as the first location for a map load/dump.

Specify High-Order Address

DOB*[f]* *ac*,**BMC**



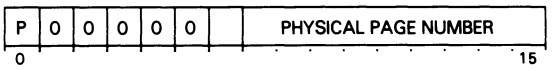
The contents of the specified accumulator determine the direction of the next map data transfer, as well as the high-order part of the physical memory address to be used. Bit 1 specifies whether map registers are to be loaded or dumped. Bits 6-15 are the the high-order 10 bits of the 20-bit physical address of the first word in memory to be transferred to or from the map. The contents of the specified accumulator are unchanged. The format of the accumulator is shown below.



BITS	NAME	CONTENTS or FUNCTION
0	---	Must be 0.
1	DUMP	When 1, the direction for the map data transfer is from the register(s) to memory.
2-5	---	Must be 0.
6-15	HI ADDR	Specify the most significant bits of the physical address for the start of the map data transfer.

Map Load Formats

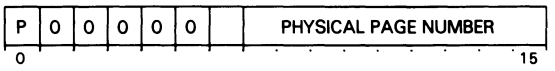
To load the map, the burst multiplexor transfers the contents of a memory buffer to the map register(s). The format of each word in the memory buffer is:



BITS	NAME	CONTENTS or FUNCTION
0	PROT	When 1, the channel cannot transfer data to/from the memory locations in the specified physical page. A transfer attempt results in a validity protect error.
1-5	---	Must be 0.
6-15	PPN	Specify the physical page number for address translation.

Map Dump Formats

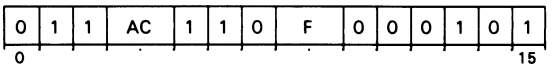
To dump the map, the burst multiplexor transfers the contents of the map register(s) to a memory buffer. The format of each word in the memory buffer is;



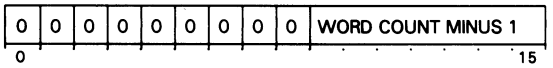
BITS	NAME	CONTENTS or FUNCTION
0	PROT	When 1, the channel cannot transfer data to/from memory in the specified physical page.
1-5	---	Reserved for future use.
6-15	PPN	Physical page number.

Specify Word Count

DOC *[f]* *ac*,BMC



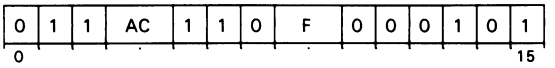
The contents of the specified accumulator determine the number of map registers to be loaded or dumped in the next map data transfer. The specified number must be one less than the number of words to be transferred. The contents of the specified accumulator are unchanged. The format of the accumulator is shown below.



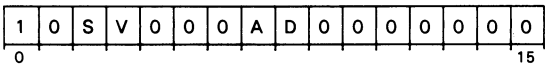
BITS	NAME	CONTENTS or FUNCTION
0-8	---	Must be 0.
9-15	COUNT	Specify a number that is one less less than the number of map registers to be loaded/dumped.

Set Status

DOC *[f]* *ac*,BMC



The contents of the specified accumulator control the diagnostic functions of the burst multiplexor. The contents of the accumulator are unchanged. The format of the accumulator is shown below.



BITS	NAME	CONTENTS or FUNCTION
0	---	Must be 1.
1	---	Must be 0.
2	S	When 1, the channel enters two-step diagnostic mode.
3	V	When 1, the channel forces a validity protect error.
4-6	---	Must be 0.
7	A	When 1, the channel forces an address parity error.
8	D	When 1, the channel forces a data parity error.
9-15	---	Must be 0.

CENTRAL PROCESSOR

Device Code - 77_8 (Primary)

Priority Mask Bit - None

Device Flag Commands

Device flag commands to the CPU determine whether the current program can be interrupted by a program interrupt request. When the interrupt enable flag is set to 1, the program can be interrupted (once the instruction following the enable has begun). When the interrupt enable flag is set to 0, the program cannot be interrupted. The CPU interrupt enable flag is controlled by the device flag commands as follows:

f=S Sets the interrupt enable flag to 1.

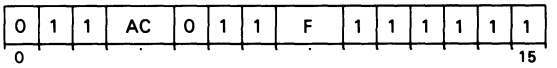
f=C Sets the interrupt enable flag to 0.

f=P If not an INTA instruction no effect. If the instruction is an INTA instruction, interprets the INTA instruction as the first word of a Vector instruction.

IORST Sets the interrupt enable flag to 0.

Interrupt Acknowledge

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



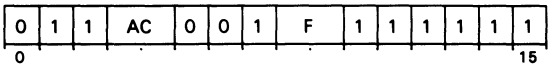
Returns device code of an interrupting device.

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.

NOTE: *The assembler recognizes the special mnemonic* **INTA ac** *to be equivalent to* **DIB ac,CPU**.

Read Switches

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



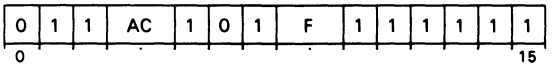
Places the contents of the console switches into an accumulator.

Places the setting of the console data switches in the specified accumulator. After the transfer, sets the Interrupt On flag according to the function specified by F.

NOTE: *The assembler recognizes the special mnemonic* **READS ac** *to be equivalent to* **DIA ac,CPU**.

Reset

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



Sends a reset signal to all devices to clear their state.

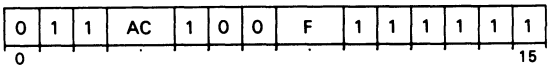
Sets the 16-bit priority mask to 0. Sets the Interrupt On flag according to the function specified by *f*.

Note that you must code an accumulator to avoid assembly errors. During execution, the accumulator field is ignored and the contents of the accumulator remain unchanged.")

NOTE: *The assembler recognizes the special mnemonic* **IORST** *to be equivalent to* **DICC 0,CPU**. *This instruction sets the Busy and Done flags as described above, and sets the Interrupt On flag to 0.*

Mask Out

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



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.

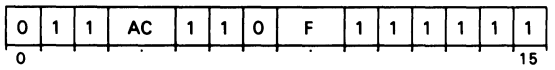
NOTE: *A 1 in any bit disables interrupt requests at devices which use that bit as a mask.*

Do not use this instruction when interrupts are enabled.

The assembler recognizes the special mnemonic **MSKO ac** *to be equivalent to* **DOB ac,CPU**.

Halt

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



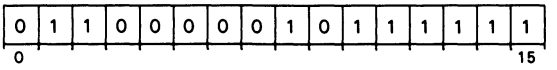
Stops the processor.

Sets the Interrupt On flag according to the function specified by F, then stops the processor. The data lights display the contents of the specified accumulator.

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

Interrupt Disable

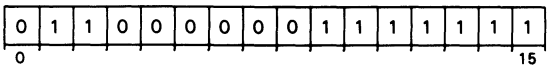
INTDS
NIOC CPU



Sets Interrupt On flag to 0.

Interrupt Enable

INTEN
NIOS CPU

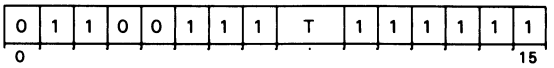


Sets Interrupt On flag to 1.

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.

CPU Skip

SKP *[t]* CPU



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

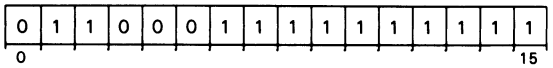
The following table lists the possible test conditions.

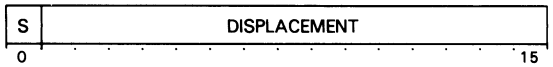
SYMBOL	VALUE	TEST
[t]=BN	00	Tests Interrupt On flag for 1
[t]=BZ	01	Tests Interrupt On flag for 0
[t]=DN	10	Tests Power Fail flag for 1
[t]=DZ	11	Tests Power Fail flag for 0

See *Programmer's Reference-Peripherals (DGC No. 014-000632)* for a complete set of examples on using the interrupt system.

Vector On Interrupting Device Code

VCT *[@]displacement[,index]*





Returns the device code of the interrupting device and uses that code as an index into a table. The value found in the table is used in one of two ways: it can be a pointer to the appropriate interrupt handler (Mode A), or as a pointer to another table (Modes B through E). This second table points to the interrupt handler and contains a new priority mask. Depending on the mode used, the instruction can also save the state of the machine by pushing various words onto the stack, create a new vector stack, and set up a priority structure.

The accompanying flow chart is a complete diagram of the operation of the Vector instruction. Note that all modes use the *vector table* to find the next address used. Mode A uses the vector table entry as the address of the interrupt handler and passes control to it immediately. Modes B through E all use the vector table address as a pointer into a *device control table (DCT)*, where the address of the interrupt handler is found, along with a new priority mask.

Three control bits determine the mode of the *Vector* instruction which will be used. Their names and locations are:

Stack Change Bit - Bit 0 of the second word of the *Vector* instruction;

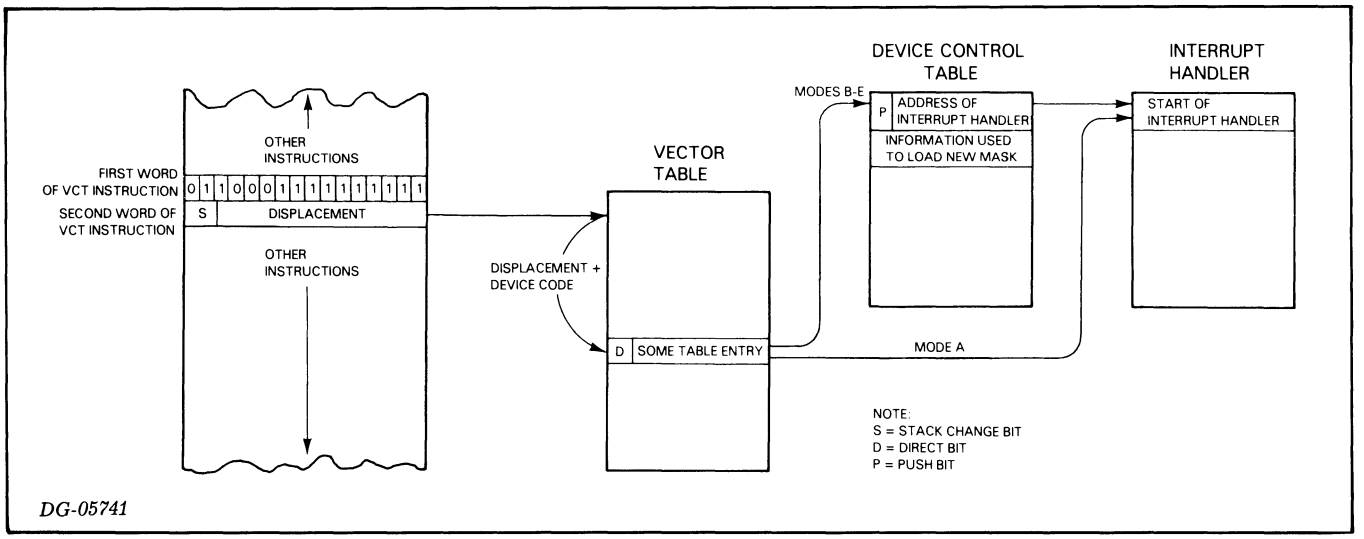
Direct Bit - Bit 0 of the selected vector table entry;

Push Bit - Bit 0 of the first word of the selected device control table.

The value of these bits collectively determine the mode of the *Vector* instruction. The bits determine the mode as follows:

DIRECT	STACK	PUSH	MODE
0	don't care	don't care	A
1	0	0	B
1	0	1	C
1	1	0	D
1	1	1	E

The diagram shows the arrangement of the various control bits and tables.



The modes perform various functions as summarized below:

MODE	FUNCTION
A	Uses device code returned by INTA as table entry to find address of interrupt handler.
B	Mode A plus: resets priority mask (saving old one) and reenables interrupts.
C	Mode B plus: pushes a normal 5-word return block (4 ACs, the program counter, and carry) onto the stack.
D	Mode B plus: sets up a new vector stack for use by the interrupt handler and saves the old stack parameters.
E	Mode C plus Mode D.

In the following paragraphs, we will consider each mode and follow through the process step-by-step.

Common Process

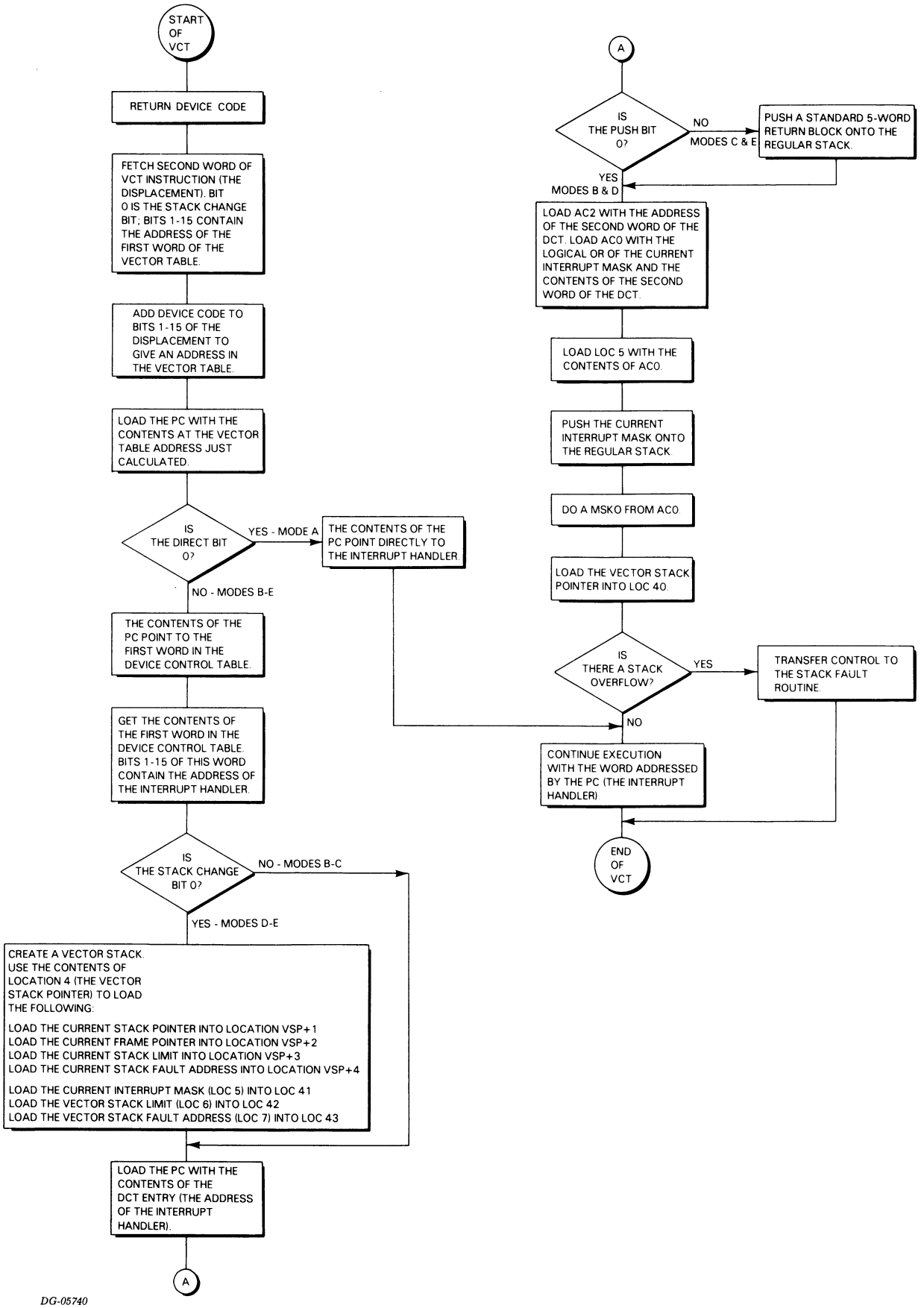
All modes perform the initial steps of the *Vector* instruction. These steps begin when the instruction returns the interrupting device code. The instruction adds the device code to the address of the start of the vector table (bits 1-15 of the second instruction word). The result is the address of an entry within the vector table. The instruction fetches the contents of this vector table entry and examines bit 0 of the entry (the direct bit).

Mode A

The instruction performs the functions of Mode A if the direct bit is 0. The values of the other control bits do not matter. In Mode A, the instruction uses bits 1-15 of the fetched vector table entry as the address of the interrupt handler of the interupting device. Control transfers immediately to the interrupt handler with all interrupts disabled.

Modes B Through E

The direct bit has the value 1 for all of these modes. The values of the push bit and the stack change bit determine which of the four modes will take place. The action of these modes can be divided into two parts: a first part, whose action varies from mode to mode; and a second part, whose action is identical for every mode. We discuss each part I separately, then the common second part.


START OF VCT
RETURN DEVICE CODE
FETCH SECOND WORD OF VCT INSTRUCTION (THE DISPLACEMENT). BIT 0 IS THE STACK CHANGE BIT; BITS 1-15 CONTAIN THE ADDRESS OF THE FIRST WORD OF THE VECTOR TABLE.
ADD DEVICE CODE TO BITS 1-15 OF THE DISPLACEMENT TO GIVE AN ADDRESS IN THE VECTOR TABLE.
LOAD THE PC WITH THE CONTENTS AT THE VECTOR TABLE ADDRESS JUST CALCULATED.
IS THE DIRECT BIT 0?
YES - MODE A
THE CONTENTS OF THE PC POINT DIRECTLY TO THE INTERRUPT HANDLER.
NO - MODES B-E
THE CONTENTS OF THE PC POINT TO THE FIRST WORD IN THE DEVICE CONTROL TABLE.
GET THE CONTENTS OF THE FIRST WORD IN THE DEVICE CONTROL TABLE. BITS 1-15 OF THIS WORD CONTAIN THE ADDRESS OF THE INTERRUPT HANDLER.
IS THE STACK CHANGE BIT 0?
NO - MODES B-C
YES - MODES D-E
CREATE A VECTOR STACK. USE THE CONTENTS OF LOCATION 4 (THE VECTOR STACK POINTER) TO LOAD THE FOLLOWING:
LOAD THE CURRENT STACK POINTER INTO LOCATION VSP+1
LOAD THE CURRENT FRAME POINTER INTO LOCATION VSP+2
LOAD THE CURRENT STACK LIMIT INTO LOCATION VSP+3
LOAD THE CURRENT STACK FAULT ADDRESS INTO LOCATION VSP+4
LOAD THE CURRENT INTERRUPT MASK (LOC 5) INTO LOC 41
LOAD THE VECTOR STACK LIMIT (LOC 6) INTO LOC 42
LOAD THE VECTOR STACK FAULT ADDRESS (LOC 7) INTO LOC 43
LOAD THE PC WITH THE CONTENTS OF THE DCT ENTRY (THE ADDRESS OF THE INTERRUPT HANDLER).
A
A
IS THE PUSH BIT 0?
NO
MODES C & E
PUSH A STANDARD 5-WORD RETURN BLOCK ONTO THE REGULAR STACK.
YES
MODES B & D
LOAD AC2 WITH THE ADDRESS OF THE SECOND WORD OF THE DCT. LOAD AC0 WITH THE LOGICAL OR OF THE CURRENT INTERRUPT MASK AND THE CONTENTS OF THE SECOND WORD OF THE DCT.
LOAD LOC 5 WITH THE CONTENTS OF AC0.
PUSH THE CURRENT INTERRUPT MASK ONTO THE REGULAR STACK.
DO A MSKO FROM AC0.
LOAD THE VECTOR STACK POINTER INTO LOC 40.
IS THERE A STACK OVERFLOW?
YES
TRANSFER CONTROL TO THE STACK FAULT ROUTINE.
NO
CONTINUE EXECUTION WITH THE WORD ADDRESSED BY THE PC (THE INTERRUPT HANDLER)
END OF VCT

DG-05740

Mode B - Part I

When the stack change bit and the push bit both have the value of 0, then Mode B takes place. The instruction uses the vector table entry as the address of the device control table *(DCT)* for the interrupting device. Bits 1-15 of the first word of the DCT contain the address of The desired interrupt handler (bit 0 is the push bit). The second word of the DCT contains information used to construct the new interrupt priority mask. Succeeding words (if any) contain information to be used by the device interrupt handler.

Mode C - Part I

When the stack change bit has the value 0 and the push bit has the value 1, then Mode C takes place. This mode performs the functions of Mode B; in addition, Mode C pushes a standard five-word return block onto the standard stack. The return block contains the contents of the four accumulators, the value of carry, and the contents of physical location 0 (the program counter return value).

Mode D - Part I

When the stack change bit has the value 1 and the push bit has the value 0, then Mode D takes place. This mode performs the functions of Mode B; in addition, Mode D sets up a new stack for the interrupt handler (using the contents of locations 4, 6, and 7) and pushes the old contents of physical locations $40\text{-}43_8$ (the user stack control words) onto the new stack.

Mode E - Part I

When the stack change bit and the push bit both have the value 1, then Mode E takes place. This mode combines the functions of modes C and D. That is, Mode E performs the functions of mode B, sets up a new stack, and pushes a 5-word return block and the old stack control words onto the new stack.

Modes B through E - Part II

Modes B through E use the same procedure for the remainder of the *Vector* instruction. During this procedure, the instruction pushes the current priority mask (location 5) onto the stack. Next, the instruction updates location 5 and performs a *Mask Out* instruction, using the logical OR of the current mask and the second word of the DCT. The instruction then sets the Interrupt On flag to 1 and passes control to the selected device interrupt handler. Note that the CPU permits one more instruction to execute (in this case, the first instruction of the interrupt handler) before the next I/O interrupt can occur.

ERCC ERROR CORRECTION

Device Code - 2_8 (Primary)

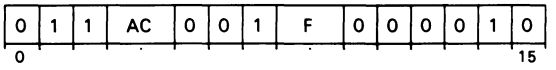
Priority Mask Bit - None

Device Flag Commands

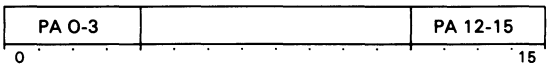
f=S	Sets the interrupt request flag and the Done flag to 0.
f=C	No effect.
f=P	No effect.
IORST	Sets the interrupt request flag, the Done flag, and the ERCC control flags (bits 14 and 15) to 0; disables error checking and correction.

Read Memory Fault Address

DIA*[f]* *ac*,**ERCC**



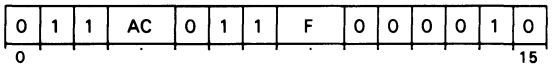
Places the complement of bits 12-15 of the physical address of the memory location in error in bits 12-15 of the specified accumulator. Places the complement of bits 0-3 of that address in bits 0-3 of the accumulator. The previous contents of the specified AC are lost. The format of the specified AC is as follows:



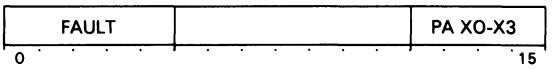
BITS	NAME	CONTENTS or FUNCTION
0-3	PA 0-3	Complement of bits 0-3 of the physical address of the memory location in error.
4-11	---	Reserved for future use.
12-15	PA 12-15	Complement of bits 12-15 of the physical address of the memory location in error.

Read Memory Fault Code

DIB *[f]* *ac*,**ERCC**



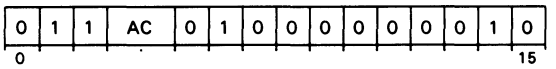
Places a 5-bit error code in bits 0-4 of the specified accumulator. This code identifies the corrected bit. Sets bits 5-11 of the accumulator to 0 and places the complement of the four high-order bits of the physical address of the failing location in bits 12-15. The accumulator format is as follows:



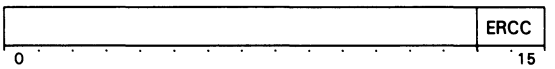
BITS	NAME	CONTENTS or FUNCTION
0-4	Code	A 5-bit code identifying which bit has an error 00000 No error 00001 Check bit 4 00010 Check bit 3 00011 Data bit 0 00100 Check bit 2 00101 Data bit 1 00110 Multiple bit error 00111 Data bit 3 01000 Check bit 1 01001 Data bit 4 01010 All 21 bits in memory are 1 01011 Data bit 6 01100 Data bit 7 01101 Data bit 8 01110 Data bit 9 01111 Multiple bit error 10000 Check bit 0 10001 Data bit 11 10010 Data bit 12 10011 Data bit 13 10100 Data bit 14 10101 All 21 bits in memory are 0 10110 Data bit 2 10111 Multiple bit error 11000 Data bit 10 11001 Multiple bit error 11010 Data bit 5 11011 Multiple bit error 11100 Data bit 15 11101 Multiple bit error 11110 Multiple bit error 11111 Multiple bit error
5-11	----	Reserved for future use.
12-15	PA X0-X3	Complement of bits X0-X3 of the physical address of the memory location in error.

Enable ERCC

DOA *[f]* *ac*,**ERCC**



Enables the ERCC option according to the setting of bits 14-15 of the specified AC. Ignores bits 0-13 of the specified AC. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-13	----	Reserved for future use
14-15	ERCC	Control the ERCC feature as follows: 00 Disable checking and correction; write valid check field. 01 Disable checking and correction; for core memory, write check field with 1111; for semiconductor memory, do not alter the check field. 10 Enable checking and correction; do not interrupt on memory error. 11 Enable checking and correction; interrupt on memory error.

Host/SP Communication

Device Code, Host to SP - 60_8 (Primary)

Priority Mask Bit, Host to SP - 5 (Primary)

Device Flag Commands

Device flag commands allow the host to interrupt the SP. These commands do this by setting interrupt request flags, busy flags, and done flags to initiate the interrupt. The SP and the host each have one busy flag, one done flag, and one interrupt request flag. The host's flags are located in and are tested by the SP. The SP's flags are located in and are tested by the host. The device flag commands for the four Host instructions are:

f=S — Sets the Host Busy and the SP interrupt request flags to 1.

f=C — Sets the SP Done and Host interrupt request flags to 0.

f=P — Sets the SP parity error and Host interrupt request flags to 0.

IORST — Sets the SP Done flag, Host interrupt request flag, and Host interrupt mask bit to 0, also resets the SP processor and its I/O devices.

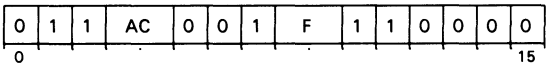
NOTE: *A SP Run flag is read as Busy by the Host. It is set to 1 when the SP is not in a halted state.*

Device Codes

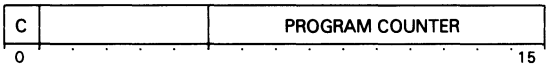
If your system contains more than one SP, you may use the mnemonics SP0, SP1, SP2, and SP3 to distinguish one SP from another. Note that the assembler does not recognize these mnemonics; therefore, when using these mnemonics in your programs, be sure you define them in a .DUSR pseudo op at the beginning of your program.

Read PC Save Register

DIA *[f]* *ac*,**SP0**



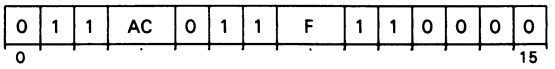
Loads the contents of the SP's PC Save register into the specified accumulator. The SP updates the PC Save register each time the SP halts. The format of the accumulator is as follows.



BITS	NAME	CONTENTS or FUNCTION
0	C	SP Carry bit
1-15	PC	SP Program counter

Read Console Buffer

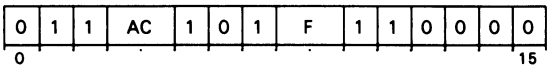
DIB *[f]* *ac*,**SP0**



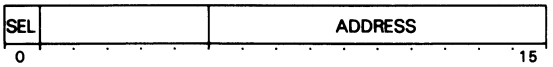
Loads the contents of the SP console buffer into the specified accumulator. The console buffer contains the result of the last console operation performed by the SP (such as *Examine*).

Read Address Buffer

DIC *[f]* *ac*,**SP0**



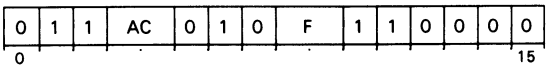
Loads the contents of the SP address buffer into the specified accumulator. The SP address buffer contains the address of the last Host memory reference. If the address save bit in the SP is 1, it will contain the address of the last memory reference to either local or host memory. The format of the register is shown below.



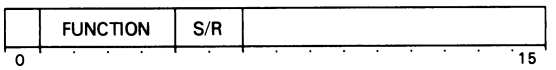
BITS	NAME	CONTENTS or FUNCTION
0	SEL	Current value of the least significant host data channel map select bit 0 DCH map A or C 1 DCH map B or D
1-15	---	SP Logical Address

Control Console Function Register

DOA *[f]* *ac*,**SP0**



Stores the contents of the specified accumulator into the SP console function register. The format of the specified accumulator is:

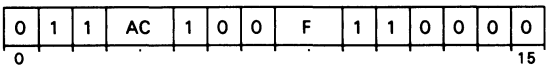


BITS	NAME	CONTENTS or FUNCTION
0	---	Reserved for future use
1-4		Selects action when bits 5 & 6 = 0: 0000 Examine AC0 0001 Examine AC1 0010 Examine AC2 0011 Examine AC3 0100 Deposit AC0 0101 Deposit AC1 0110 Deposit AC2 0111 Deposit AC3 1000 Deposit 1001 Deposit Next 1010 Examine 1011 Examine Next 1100 Start 1101 Execute 1110 Program Load 1111 Continue
5,6	S/R	11 No action 10 STOP 01 RESET 00 bits 1-4 specify action
7-15	---	Reserved for future use.

NOTE: *Select the* Inst *function by specifying* **STOP** *and* **CONTINUE** *together (i.e., bits 1-6 = 111110$_2$). The SP ignores functions other than* **RESET**, **STOP**, **EXAMINE**, *and* **INST** *when it is running. The* **STOP** *function halts the SP* after *it completes the instruction currently executing.*

Control Switch Register

DOB*[f]* *ac*,**SP0**



Stores the contents of the specified accumulator into the SP switch register. The SP switch register contains the address or data for the SP console operations.

MEMORY ALLOCATION and PROTECTION

Device Code - 3_8 (Primary)

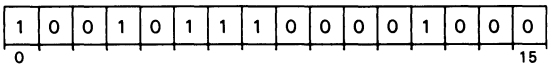
Priority Mask Bit - None

Device Flag Commands

f=S	No effect.
f=C	No effect.
f=P	Enables Map Single Cycle.
IORST	Disables Map.

Load Map

LMP



Under control of AC1 and AC2, loads successive words from memory into the MAP where they are used to define a user or data channel map.

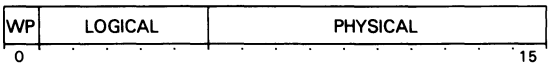
AC1 must contain an unsigned integer, which is the number of words to be loaded into the MAP. Bits 1-15 of AC2 must contain the address of the first word to be loaded. If bit 0 of AC2 is 1, the instruction follows the indirection chain and places the resultant effective address in AC2. AC0 and AC3 are ignored and their contents remain unchanged.

For each word loaded, the instruction decrements the count in AC1 by one and increments the source address in AC2 by 1. Upon completion of the instruction, AC1 contains 0, and AC2 contains the address of the word following the last word loaded.

This instruction is interruptible in the same manner as the *Block add and move* instruction. If you issue this instruction while in mapped mode, with I/O protection enabled, the map will not be altered. AC1 and AC2 will be used and their contents modified as described above. No I/O trap will occur.

The words loaded into the MAP define the address translation functions for the various user and data channel maps. The contents of the MAP field (bits 6-8) of the MAP status register determine which map is affected by the *Load map* instruction. You can alter this field using either the *Load map status* or the *Initiate page check* instruction.

The format of the words loaded into the MAP is as follows:

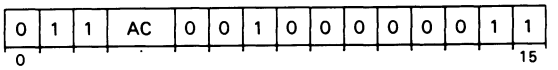


BITS	NAME	CONTENTS or FUNCTION
0	WP	Unused for data channel maps; write protect for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

NOTE: *Declare a logical page invalid by setting the Write Protect bit to 1 and all of bits 6-15 to 1.*

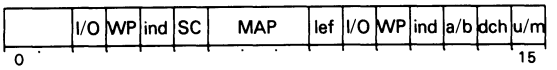
Read Map Status

DIA*[f]* *ac*,MAP



Reads the status of the current map.

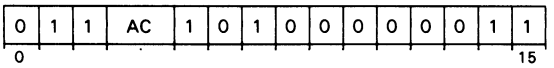
Places the contents of the MAP status register in the specified AC. The previous contents of the specified AC are lost. The format of the information placed in the specified AC is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-1	---	Reserved for future use.
2	I/O	If 1, the last protection fault was an I/O protection fault.
3	WP	If 1, the last protection fault was a write protection fault.
4	IND	If 1, the last protection fault was an indirect protection fault.
5	Single Cycle	If 1, the last map reference was a *Map Single Cycle* instruction.
6-8	Map	Indicates which map will be loaded by next *Load map* instruction as follows: 000 User A 001 Reserved for future use 010 User B 011 Reserved for future use 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
9	LEF	If 1, the *Load Effective Address* instruction was enabled by the last *Load Map Status* instruction.
10	I/O	If 1, I/O protection was enabled by the last *Load Map Status* instruction.
11	WP	If 1, write protection was enabled by the last *Load Map Status* instruction
12	IND	If 1, indirect protection was enabled by the last *Load Map Status* instruction.
13	A/B	If 0, the last *Load Map Status* instruction enabled map A. If 1, the last *Load Map Status* instruction enabled user map B.
14	DCH Enable	If 1, the mapping of the data channel addresses is enabled.
15	User Mode	If 1, the last I/O interrupt occurred while in user mode.

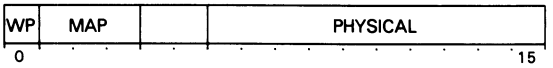
Page Check

DIC *ac*,MAP



Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.

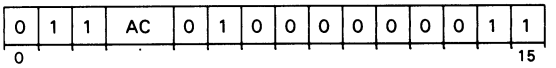
Places the number of the physical page which corresponds to the logical page specified by the preceding *Initiate Page Check* or *Load Map Status* instruction in bits 6-15 of the specified AC. Places additional information about this page in bits 0-3 and destroys the previous contents of the AC. The format of the information placed in the specified AC is as follows:



BITS	NAME	CONTENTS or FUNCTION
0	WP	The write protect bit for the logical page which corresponds to the physical page specified by bits 6-15.
1-3	Map	The map which was used to perform the translation between logical page number and physical page number is as follows: 000 User A 001 Reserved for future use. 010 User B 011 Reserved for future use. 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
4-5	---	Reserved for future use.
6-15	Physical	The number of the physical page which corresponds to the logical page given in the preceding INITIATE PAGE CHECK instruction. If all these bits are 1, and WP (bit 0) is 1, then the logical page is validity protected.

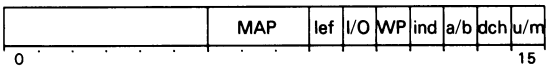
Load Map Status

DOA *ac*,MAP



Defines the parameters of a new map.

Places the contents of the specified AC in the MAP status register. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:

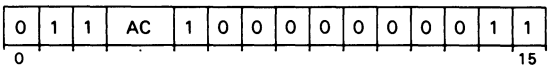


BITS	NAME	CONTENTS or FUNCTION
0-5	---	Reserved for future use.
6-8	MAP SEL	Specify which map will be loaded by the next Load Map instruction as follows: 000 User A 001 Reserved for future use 010 User B 011 Reserved for future use 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
9	LEF	If 1, the Load Effective Address instruction will be enabled for the next user
10	I/O	If 1, I/O protection will be enabled for the next user
11	WP	If 1, write protection will be enabled for the next user
12	IND	If 1, indirect protection will be enabled for the next user
13	A/B	If 0, the next user map enabled will be that for user A If 1, the next user map enabled will be that for user B
14	DCH Enable	If 1, the mapping of data channel addresses will be enabled immediately after this instruction
15	User Mode	If 1, mapping of CPU addresses will commence with the first memory reference *after* the next *indirect* reference or return type instruction **(POPB, POPJ, RTN, RSTR)**

NOTE: *If the* Load Map Status *instruction sets the User Enable bit to 1, this inhibits the interrupt system and the MAP waits for either an indirect reference or return type instruction. Either event releases the interrupt system and allows the MAP to begin translating addresses (using the user map specified by bit 13 of the MAP status register). Address translation resumes (1) after the first level of the next indirect reference; or (2) after the first* Pop Block, Pop Jump, Return, *instruction that does not cause a stack fault.*

Map Page 31

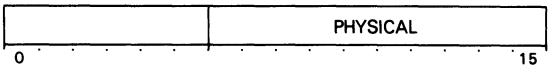
DOB *ac*,MAP



Specifies that mapping take place for a single page of an unmapped address space. Mapping is always done for locations 76000_8 through 77777_8 (logical page 31). This is the only page which can be mapped when in unmapped address space. You can use this instruction to access a page of a user's memory space when in unmapped mode.

Bits 6-15 of the specified AC are transferred to the MAP. These bits specify a physical page number to which logical page 31 will be mapped when in the unmapped mode.

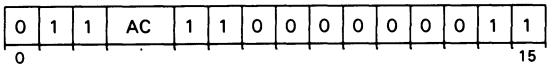
The contents of the specified AC remain unchanged. The format of the specified AC is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-5	---	Reserved for future use.
6-15	Physical	The number of the physical page to which logical page 31 should be mapped when in unmapped mode.

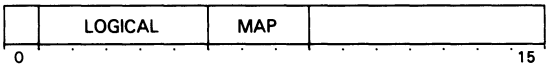
Initiate Page Check

DOC *ac*,MAP



Identifies a logical page. The *Page Check* instruction will find the corresponding physical page.

Transtransfers the contents of the specified AC to the MAP for later use by the *Page Check* or *Load Map* instruction. Leaves the contents of the specified AC unchanged. The format of the specified AC is as follows:

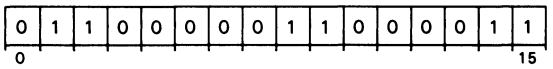


BITS	NAME	CONTENTS or FUNCTION
0	---	Reserved for future use.
1-5	Logical Page	Number of the logical block for which the check is requested.
6-8	Map	Specify which map should be used for the check as follows: 000 User A 001 Reserved for future use 010 User B 011 Reserved for future use 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
9-15	---	Reserved for future use.

Map Single Cycle

Disable User Mode

NIOP *ac*,**MAP**



Issued from unmapped mode, the instruction maps one memory reference using the last user map; issued from User mode with LEF mode and I/O protection disabled, the instruction simply turns off the map, returning it to unmapped mode. It is used by the supervisor to access a user's memory space when only one or two references are required. It is also used by a privileged user to turn off memory mapping.

From unmapped mode - Enables the user map for one memory reference. Maps the first memory reference of the next **LDA, ELDA, STA** or **ESTA** instruction. After the memory cycle is mapped, the instruction disables the user map.

> NOTE: *The interrupt system is disabled from the beginning of the* Map single cycle *instruction until after the next* LDA, ELDA, STA *or* ESTA *instruction.*

From User mode - If **LEF** Mode and I/O protection is disabled, this instruction turns off the MAP. All subsequent memory references are unmapped until the map is reactivated with a *Load map status* instruction.

PROGRAMMABLE INTERVAL TIMER

Device Code - 43_8 (Primary)

Priority Mask Bit - 11

Device Flag Commands

f=S — Sets the Busy flag to 1 and the Done flag and interrupt request flag to 0; begins the counting cycle.

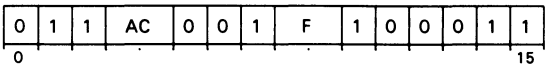
f=C — Sets the Busy and Done flags and the interrupt request flag to 0; stops the counting cycle.

f=P — No effect.

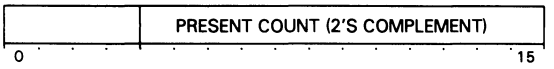
IORST — Sets the Busy and Done flags, the interrupt request flag, the initial count register, the count output buffer, and the interrupt mask bit (bit 11) to 0; stops the counting cycle.

Read Count

DIA *[f]* *ac*,**PIT**



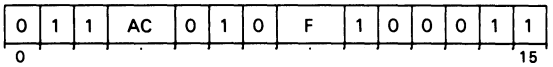
Places the value of the Programmable Interval Timer's Counter in bits 0-15 of the specified accumulator destroying the accumulator's previous contents. After the data transfer, performs the function specified by F. The format of the specified accumulator is as follows:



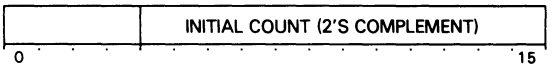
BITS	NAME	CONTENTS or FUNCTION
0-15	Count	Current value of the PIT counter within one count cycle.

Specify Initial Count

DOA*[f]* *ac*,**PIT**



Loads bits 0-15 of the specified accumulator into the Programmable Interval Timer's Initial Count Register. After the data transfer, performs the function specified by F. The contents of the specified accumulator remain unchanged; the format of the accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-15	Initial Count	Two's complement of the number of 100 microsecond intervals between interrupts

REAL TIME CLOCK

Device Code - 14_8 (Primary)

Priority Mask Bit - 13

Device Flag Commands

f=S Sets the Busy flag to 1, and the Done flag and interrupt request flag to 0; enables RTC interrupts.

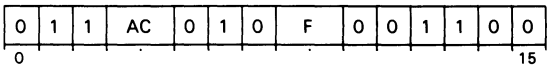
f=C Sets the Busy and Done flags and the interrupt request flag to 0; disables RTC interrupts.

f=P No effect.

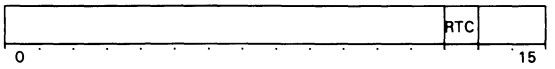
IORST Sets the Busy and Done flags, the interrupt request flag, the interrupt mask bit (bit 13), and the clock frequency select bits to 0; disables RTC interrupts.

Select RTC Frequency

DOA*[f]* *ac*,**RTC**



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:



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

PRIMARY ASYNCHRONOUS LINE INPUT

Device Code - 10_8 (Primary)

Priority Mask Bit - 14

Device Flag Commands

f=S Sets the Busy flag to 1 and the Done flag to 0.

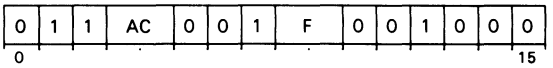
f=C Sets the Busy and Done flags and the interrupt request flag to 0.

f=P No effect.

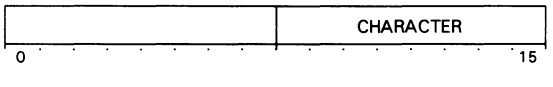
IORST Sets the Busy and Done flags, the interrupt request flag, and the interrupt mask bit (bit 14) to 0.

Read Character Buffer

DIA *[f] ac*,**TTI**



Places the contents of the controller's input buffer in bits 8-15 of the specified accumulator. After the data transfer, sets the controller's Busy and Done flags according to the function specified by F. The format of the specified accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-7	----	Reserved for future use.
8-15	Char-acter	The 8 bit character or 7 bit character with parity in bit position 8 read from the input buffer.

PRIMARY ASYNCHRONOUS LINE OUTPUT

Device Code - 11_8 (Primary)

Priority Mask Bit - 15

Device Flag Commands

f=S Sets the Busy flag to 1 and the Done flag to 0; begins transmission of the character contained in the output buffer.

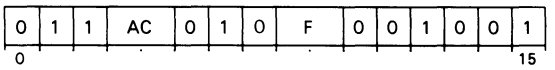
f=C Sets the Busy and Done flags and the interrupt request flag to 0.

f=P No effect.

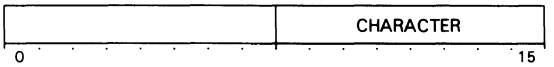
IORST Sets the Busy and Done flags, the interrupt request flag, and the interrupt mask bit (bit 15) to 0.

Load Character Buffer

DOA *[f] ac*,**TTO**



Loads bits 8-15 of the specified accumulator into the controller's output buffer. After the data transfer, sets the controller's Busy and Done flags according to the function specified by F. The contents of the specified accumulator remain unchanged. The format of the specified accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-7	----	Reserved for future use.
8-15	DATA	The 8-bit character or 7-bit character with parity in bit position 8 to be placed in the output buffer.

Chapter VI
SP Communication Instructions

This chapter lists the SP instructions used to communicate with the host. For each instruction we include:

- the mnemonic recognized by the assembler
- the bit format required
- the format of any arguments involved
- the functional description of each instruction

CODING AIDS

We use certain conventions and abbreviations in 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
f	Device flag command
AC	Accumulator

SP/Host Communication

Device Code, SP to Host and local I/O - 4_8 (Primary)

Priority Mask Bit, SP to Host - 5 (Primary)

Priority Mask Bit, SP Map - None

Device Flag Commands

Device flag commands allow the SP to interrupt the host. These commands do this by setting interrupt request flags, busy flags, and done flags to initiate the interrupt. The SP and the host each have one busy flag, one done flag, and one interrupt request flag. The host's flags are located in and are tested by the SP. The SP's flags are located in and are tested by the host. The device flag commands for the four SP instructions are:

f=S Sets the SP Done and Host interrupt request flags to 1.

f=C Sets the Host Busy and SP interrupt request flags to 0.

f=P Sets the Host Done and SP interrupt request flags to 0.

IORST Sets bits 2-4, 14 and 15 of the map status and parity control register, Host Done flag, SP interrupt request flag, and SP interrupt mask bits to 0.

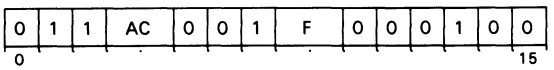
NOTE: *A SP Run flag is read as Busy by the Host. It is set to 1 when the SP is not in a halted state.*

Device Codes

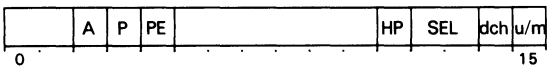
If your system contains more than one SP, you may use the mnemonics SP0, SP1, SP2, and SP3 to distinguish one SP from another. Note that the assembler does not recognize these mnemonics; therefore, when using these mnemonics in your programs, be sure you define them in a **.DUSR** pseudo op at the beginning of your program.

Read Map Status and Parity Control

DIA *[f]* *ac*,**ISP0**



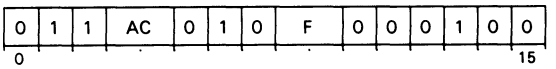
Loads the specified accumulator with the map status and parity control bits, as well as the MAP host/local flag selected by the previous **DOA** instruction. The format of the accumulator is shown below:



BITS	NAME	CONTENTS or FUNCTION
0,1	---	Reserved for future use.
2	A	Address Save on
3	P	Parity Test on
3	PE	Parity Checking on
5-10	---	Reserved for future use.
11	HP	Bit for MAP page selected by previous **DOA**: 0 = page is mapped into SPU local memory. 1 = page is mapped into host memory.
12,13	SEL	Currently selected host data channel: 00 DCH Map A 01 DCH Map B 10 DCH Map C 11 DCH Map D
14	DCH	Current state of DCH MODE.
15	UM	Current state of USER MODE.

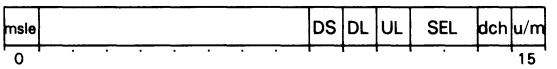
Control (and Select) Map and Page/Parity

DOA *[f]* *ac*,**ISP0**

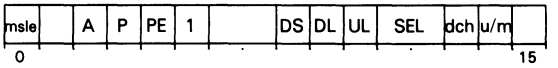


Stores the specified accumulator into the map status and parity control register as summarized below. Depending on the contents of bit 0 in the specified accumulator, the contents of bits 12-15 may be ignored; also, the contents of bit 6 controls the interpretation of bits 1-5. Consequently, the accumulator may take one of the following formats:

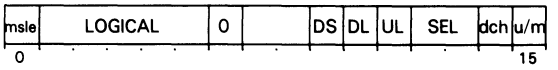
Format 1, Control Map Only



Format 2, Control Map and Parity



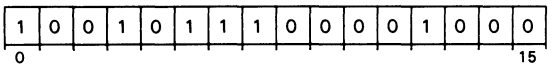
Format 3, Control Map and Select Page



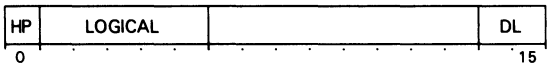
BITS	NAME	CONTENTS or FUNCTION
0	MSLE	Permit loading of map status (bits 12-15).
1	---	Used for map reading only.
2	A	Address Save on.
3	P	Parity Test on.
4	PE	Parity checking on.
1-5	LOGICAL	Selects Logical Page
6	PCL	Permit loading of parity control flags.
7-8	---	Reserved for future use.
9	DS	Suppress loading of DCH map select (bits 12,13).
10	DL	Suppress loading of DCH mode (bit 14).
11	UL	Suppress loading of USER mode (bit 15).
12,13	SEL	Selects host data channel map for references to host memory: 00 DCH A 01 DCH B 10 DCH C 11 DCH D
14	DCH	DCH mode on.
15	U/M	USER mode on.

Load Map

LMP



Load a number of map entries from a table in memory. AC2 must contain the starting address of the table. AC1 must contain the number of entries to load. AC0 and AC3 are unused. The format of each entry to be loaded into the SP map is shown below.



BITS	NAME	FUNCTION
0	HP	0 = map page into local memory. 1 = map page into host memory.
1-5	LOGICAL	Logical page number.
6-14	---	Reserved for future use.
15	DL LD	0 = load entry into USER map. 1 = load entry into DCH map.

Chapter VII
INTEGRAL ARRAY PROCESSOR INSTRUCTIONS

This chapter lists all the array processing instructions for the S/250. We have arranged the instructions in alphabetical order according to mnemonics as recognized by the assembler.

For each instruction we include:

- the mnemonic recognized by the assembler
- the bit format required
- a mathematical formula describing the operation of the instruction
- a functional description of each instruction
- the contents and format of the parameter block
- a list of the relevant status flags.

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:

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.).

NOTATIONAL CONVENTIONS

The tables below list the meanings of the various symbols which are used throughout this chapter.

OPERATIONS

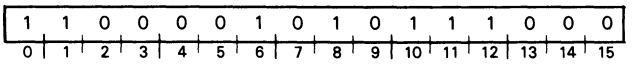
Symbol	Meaning
+	addition
-	subtraction
*	multiplication
/	division
'	bit-reversal
||	absolute value
Rel]	real part
Iml]	imaginary part
Σ	summation of terms
Π	product of terms
$<$	less than
=	equal
$>$	greater than
$\leqslant$	less or equal
$\neq$	not equal
$\geqslant$	greater or equal
$\leftarrow$	value assignment

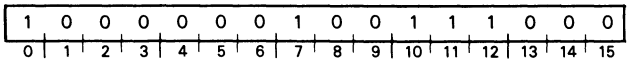
VARIABLES

Symbol	Meaning
X	Source array.
Y	Source array.
Z	Destination array.
M	Main memory operand; may be source or destination, array or scalar depending on instruction.
W	Working register.
P	Scratchpad.
Q	Temporary intermediate array (transparent to the user).
S_a	Step register for array "a", where "a" is any array (X,Y,Z, or M).

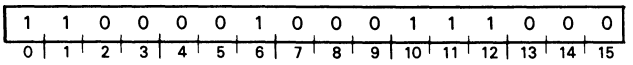
Add Real Arrays

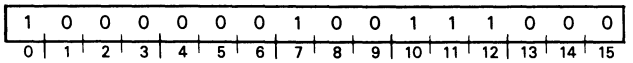
ARA





ARAP





$Z_{(n\ S_Z)} \leftarrow X_{(n\ S_X)} + Y_{(n\ S_Y)}$

$n = 0, 1, \ldots, N\text{-}1$

ACTION - Adds each element of the source array X to the corresponding element of the source array Y and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

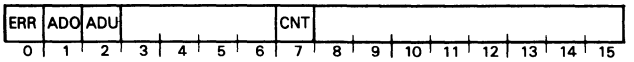
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

*ARA only.

PARAMETER BLOCK FORMAT

0	ERROR MASK	
1	@	ERROR TRAP ADDRESS
2		ELEMENT COUNT
3		Z_0 INDEX
4		X_0 INDEX
5		Y_0 INDEX
6		
7		
8	CONTINUATION REGISTER	
9		Z STEP VALUE
10		X STEP VALUE
11		Y STEP VALUE
12		
13		
14		
15		

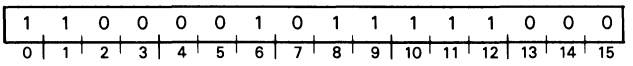
STATUS FLAGS UPDATED

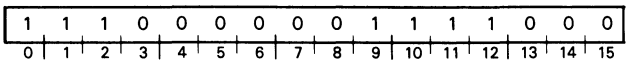


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
7	CNT	Illegal element count.

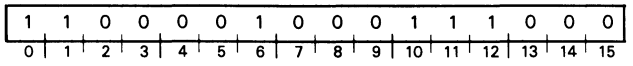
Add Real Scalar to Array

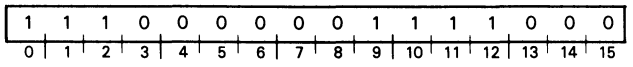
ARS





ARSP





$Z_{(n\ S_Z)} \leftarrow X_{(n\ S_X)} + W$

$n = 0, 1, \ldots, N\text{-}1$

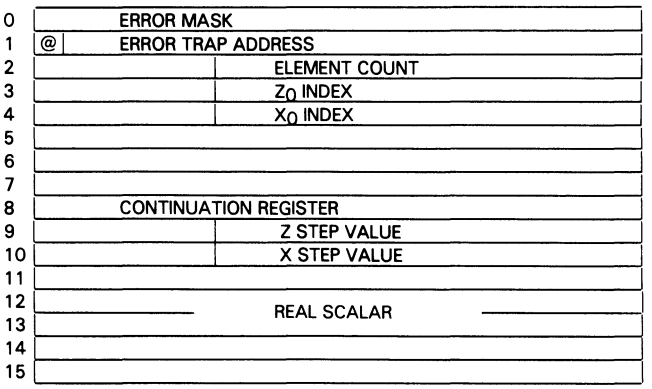
ACTION - Places the real scalar specified in the parameter block in the working register.* Then the instruction adds the contents of the working register to each element of the real source array X, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

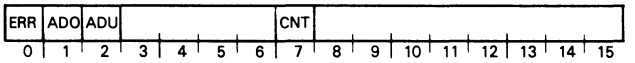
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
12,13 *	W	Real scalar to be placed in working register.

*ARS only.

PARAMETER BLOCK FORMAT



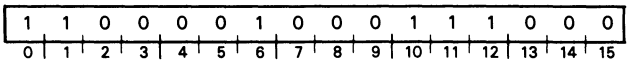
STATUS FLAGS UPDATED

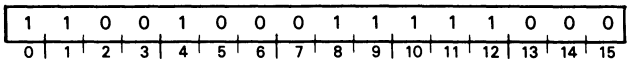


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
7	CNT	Illegal element count.

Bit-reverse Indices of Complex Array

BRC





$X_n \leftarrow X_{n'}$,

$n = 0, 1, \ldots, N-1$

where

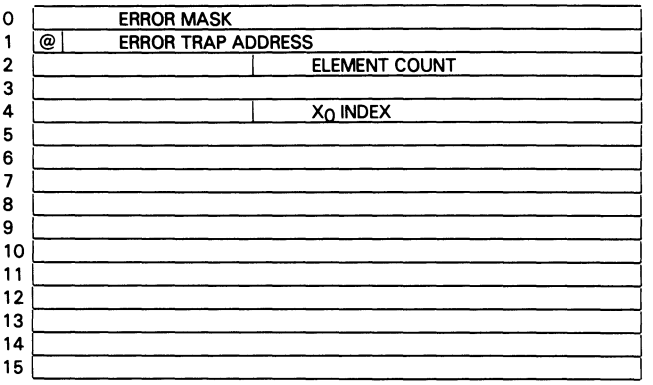
n' = Bit reverse of n within $\log_2 N$ bits.

ACTION - Rearranges the elements of the array X such that their indices are bit-reversed with respect to their original contents. For example, if you executed **BRC** on a 16-element array, X_1 would be exchanged with X_8, since 1 (0001_2) is the bit-reverse of 8 (1000_2). Similarly, X_2 would be exchanged with X_4, etc. The format of the parameter block is as follows:

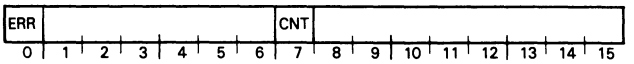
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count: must be a power of 2 less than or equal to 1024.
4	X_0 INDEX	Index of complex array X.

PARAMETER BLOCK FORMAT



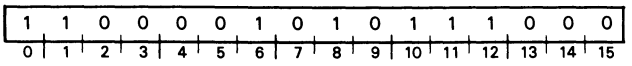
STATUS FLAGS UPDATED

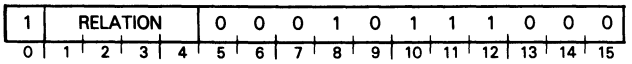


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

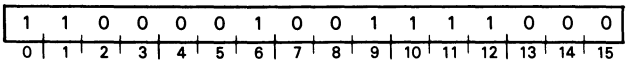
Compare Arrays

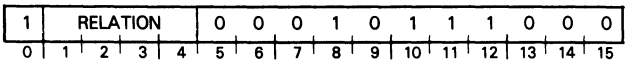
CMA *rel*





CMAP *rel*





Evaluates, over n, the expression:

$$X_{(n\,S_X)}\ \ rel\ \ Y_{(n\,S_Y)}$$

$n = 0, 1, \ldots, N\text{-}1$

Initially:

$i \leftarrow 1;$

For each n where the expression is true:

$M_0 \leftarrow i,$

$M_i \leftarrow n\,S_x$

and

$i \leftarrow i + 1$

Relational operators:

rel	Meaning	Bit field
LT	Less Than	0000
LE	Less or Equal	0001
EQ	Equal	0010
NE	Not Equal	0011
GT	Greater Than	0100
GE	Greater or Equal	0101
A LT	Absolute value Less Than	0110
A LE	Absolute value Less or Equal	0111
A EQ	Absolute value Equal	1000
A NE	Absolute value Not Equal	1001
A GT	Absolute value Greater Than	1010
A GE	Absolute value Greater or Equal	1011

ACTION - Compares corresponding elements of the two source arrays X and Y. The first word of the integer result array M contains the number of elements for which the relation specified by *rel* is true. Each subsequent word of M contains a number indicating one of the elements of X for which the relation was true. The numbers are relative to the start of X, and are multiplied by the current contents of the X step register. Note that M may be anywhere in main memory.

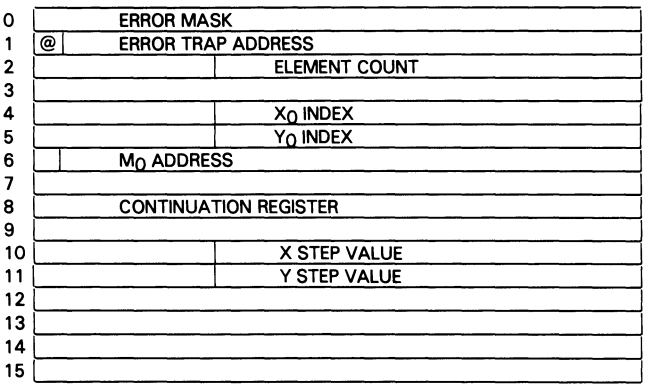
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

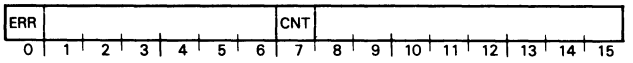
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
6	M_0 ADDR	Main memory address of integer array M.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

*CMA only.

PARAMETER BLOCK FORMAT

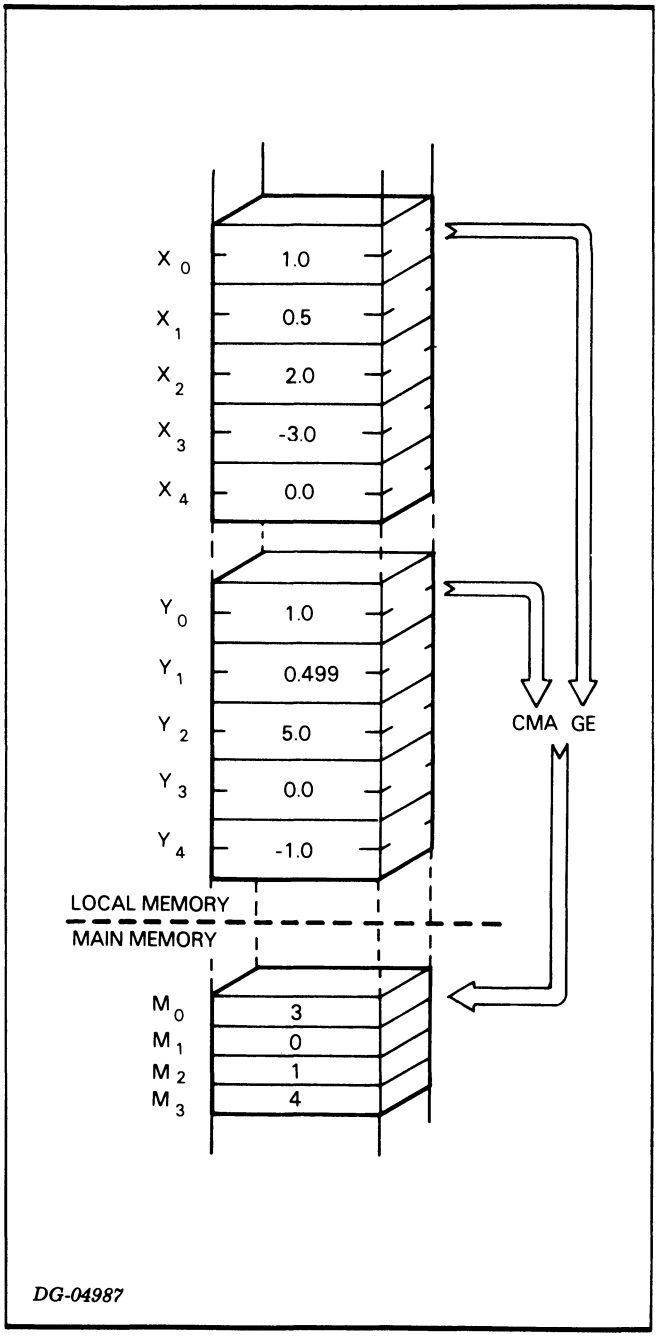


STATUS FLAGS UPDATED



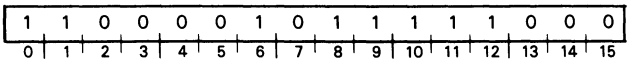
BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

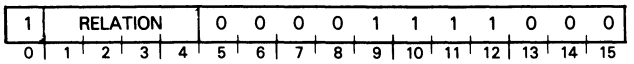
The accompanying diagram shows the result of a typical execution of CMA on two 5-element arrays. M_0 indicates that there are 3 elements in X which are *G*reater than or *E*qual to the corresponding elements of Y. M_1 through M_3 are the subscripts of the element pairs for which the relation is true.



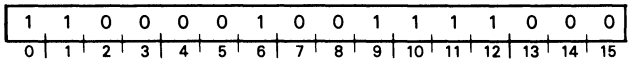
Compare Real Scalar to Array

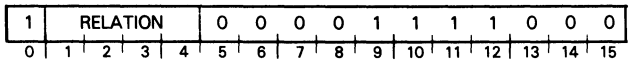
CMS *rel*





CMSP *rel*





Evaluates, over n, the expression:

$X_{(n\,S_X)}$ *rel* W

$n = 0, 1, \ldots, N\text{-}1$

Initially:

$i \leftarrow 1;$

For each n where the expression is true:

$M_0 \leftarrow i,$

$M_i \leftarrow n\,S_x$

and

$i \leftarrow i + 1$

Relational operators:

rel	Meaning	Bit field
LT	Less Than	0000
LE	Less or Equal	0001
EQ	Equal	0010
NE	Not Equal	0011
GT	Greater Than	0100
GE	Greater or Equal	0101
A LT	Absolute value Less Than	0110
A LE	Absolute value Less or Equal	0111
A EQ	Absolute value Equal	1000
A NE	Absolute value Not Equal	1001
A GT	Absolute value Greater Than	1010
A GE	Absolute value Greater or Equal	1011

ACTION - Places the real scalar from the parameter block in the working register.* Then the instruction compares the scalar in the working register to each element of the source array X. The 0th word of the integer result array M contains the number of elements for which the relation *"rel"* is true. Each subsequent word in M contains a number indicating one of the elements of X for which the relation was true. The numbers are relative to the start of the array, and are multiplied by the current value of the X step value.

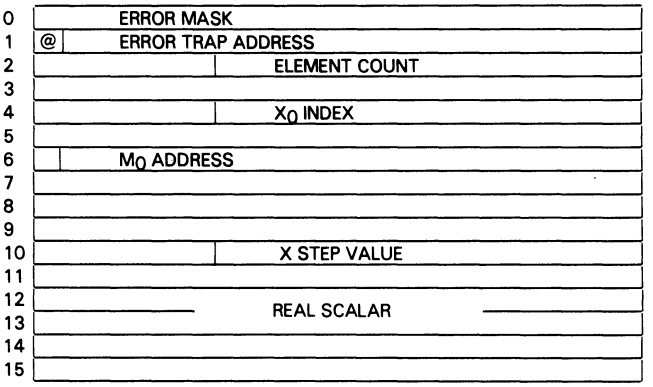
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

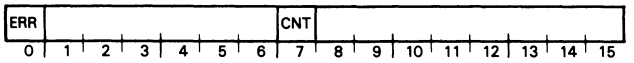
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
6	M_0 ADDR	Main memory address of integer array M.
10 *	S_X	Step value for array X.
12,13 *	W	Real scalar to be placed in working register.

*CMS only.

PARAMETER BLOCK FORMAT



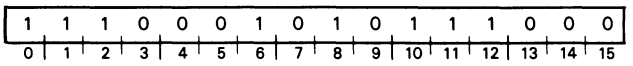
STATUS FLAGS UPDATED

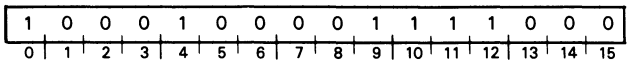


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

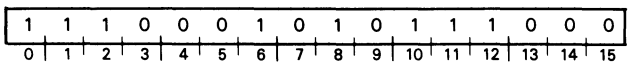
Convolution of Complex Arrays

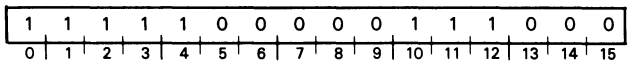
CONC





CONCZ





$$Z_n \leftarrow \sum_{m=0}^{M-1} X_{n-m} Y_m$$

n = 0, 1, . . . , N-1

ACTION - Computes the convolution product of the source arrays X and Y, and places the result in the destination array Z.

CONC assumes that X has length N + M - 1. The first M - 1 elements are the initial conditions. They are stored in elements with negative subscripts, $X_{(-M+1)}$ through X_{-1}.

CONCZ assumes that all initial conditions are zero. The number of elements in X is equal to N.

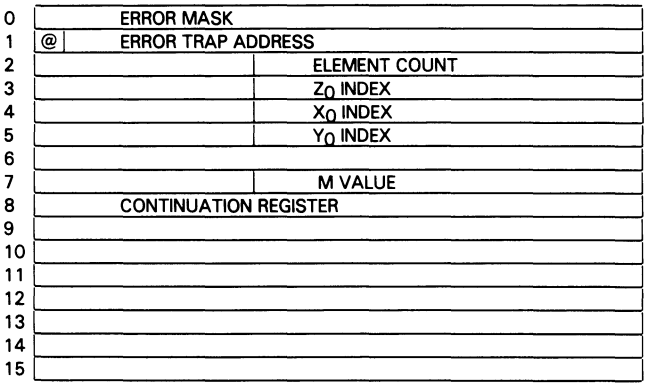
> NOTE: *The sizes of the source arrays are limited by the condition that 2 * (M + N) - 1 must be less than or equal to 1024.*

The format of the parameter block is as follows:

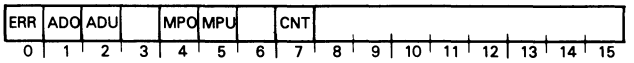
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count for source array X.
3	Z_0 INDEX	Index of complex array Z.
4	X_0 INDEX	Index of complex array X.
5	Y_0 INDEX	Index of complex array Y.
7	M	Element count for source array Y.
8	---	Continuation register.

PARAMETER BLOCK FORMAT



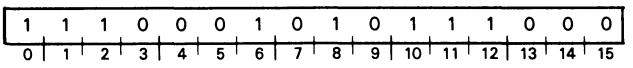
STATUS FLAGS UPDATED

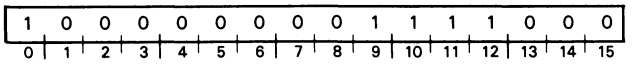


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

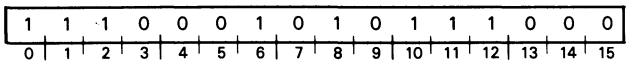
Convolution of Real Arrays

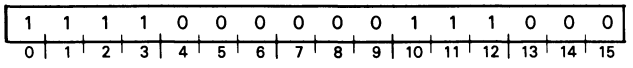
CONR





CONRZ





$$Z_n \leftarrow \sum_{m=0}^{M-1} X_{n-m} Y_m$$

n = 0, 1, . . . , N-1

ACTION - Computes the convolution product of the source arrays X and Y, and places the result in the destination array Z.

CONR assumes that X has length N + M - 1. The first M - 1 elements are the initial conditions. They are stored in elements with negative subscripts, $X_{(-M+1)}$ through X_{-1}.

CONRZ assumes that all initial conditions are zero. The number of elements in X is equal to N.

NOTE: *The sizes of the source arrays X and Y are limited by the condition that 2 * (M + N) - 1 must be less than or equal to 2048.*

The format of the parameter block is as follows:

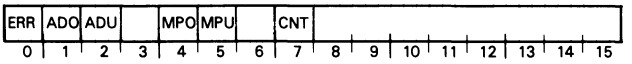
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count for source array X.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
7	M	Element count for source array Y.
8	---	Continuation register.

PARAMETER BLOCK FORMAT

0	ERROR MASK	
1	@	ERROR TRAP ADDRESS
2		ELEMENT COUNT
3		Z_0 INDEX
4		X_0 INDEX
5		Y_0 INDEX
6		
7		M VALUE
8	CONTINUATION REGISTER	
9		
10		
11		
12		
13		
14		
15		

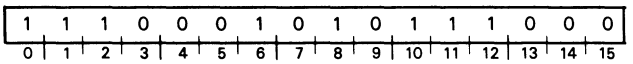
STATUS FLAGS UPDATED

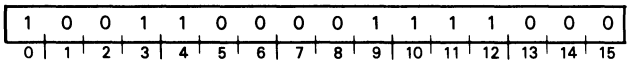


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

Correlation of Complex Arrays

CORC





$$Z_n \leftarrow \sum_{m=0}^{M-1} X_{n+m} Y_m$$

$n = 0, 1, \ldots, N-1$

ACTION - Computes the correlation product of the source arrays X and Y, and places the result in the destination array Z. The length of X is assumed to be M + N - 1.

> **NOTE:** *The sizes of the source arrays X and Y are limited by the condition that 2 * (M + N) - 1 is less than or equal to 1024.*

The format of the parameter block is as follows:

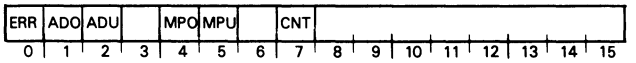
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count N used to compute length of source array X.
3	Z_0 INDEX	Index of complex array Z.
4	X_0 INDEX	Index of complex array X.
5	Y_0 INDEX	Index of complex array Y.
7	M	Element count M for source array Y.
8	---	Continuation register.

PARAMETER BLOCK FORMAT

0	ERROR MASK	
1	@ ERROR TRAP ADDRESS	
2		ELEMENT COUNT
3		Z_0 INDEX
4		X_0 INDEX
5		Y_0 INDEX
6		
7		M VALUE
8	CONTINUATION REGISTER	
9		
10		
11		
12		
13		
14		
15		

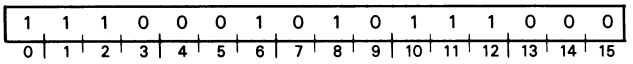
STATUS FLAGS UPDATED

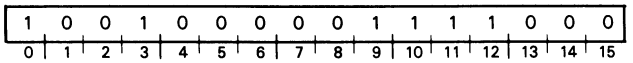


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

Correlation of Real Arrays

CORR





$$Z_n \leftarrow \sum_{m=0}^{M-1} X_{n+m} Y_m$$

n = 0, 1, . . . , N-1

ACTION - Computes the correlation product of the source arrays X and Y, and places the result in the destination array Z. The length of X is assumed to be N + M - 1.

> **NOTE:** *The sizes of the source arrays X and Y are limited by the condition that 2 * (M + N) - 1 is less than or equal to 2048.*

The format of the parameter block is as follows:

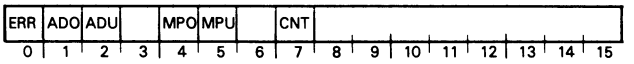
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count N used to compute length of source array X.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
7	M	Element count M for source array Y.
8	---	Continuation register.

PARAMETER BLOCK FORMAT

0		ERROR MASK	
1	@	ERROR TRAP ADDRESS	
2			ELEMENT COUNT
3			Z_0 INDEX
4			X_0 INDEX
5			Y_0 INDEX
6			
7			M VALUE
8		CONTINUATION REGISTER	
9			
10			
11			
12			
13			
14			
15			

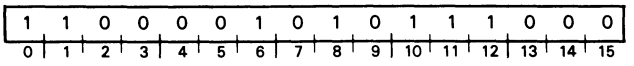
STATUS FLAGS UPDATED

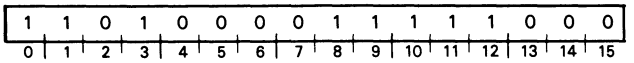


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

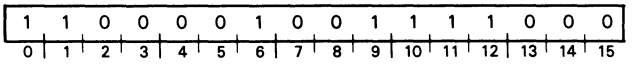
Create a Real Array

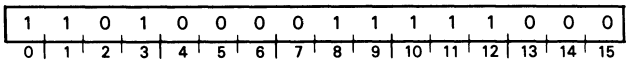
CRE





CREP





$Z_{(n-1)\,S_Z} \leftarrow X_{(M_n)}$

$n = 1, 2, \ldots, N$

where

$N = M_0$

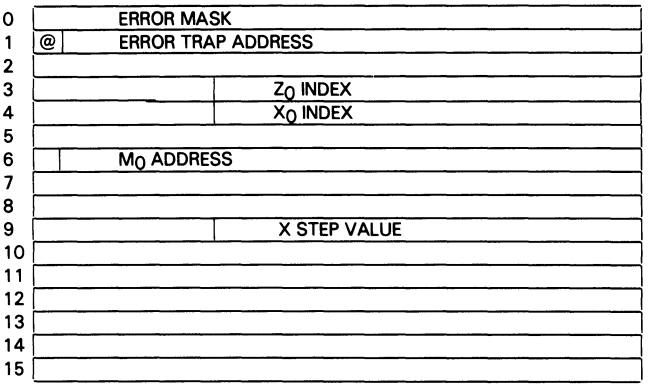
ACTION - Creates a real array Z by loading it with elements of the source array X as selected by the indices in the integer array M. The 0th word of M contains its length. Note that M may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

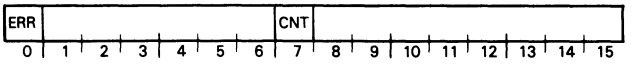
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
6	M_0 ADDR	Main memory address of integer array M.
9 *	S_Z	Step value for array Z.

*CRE only.

PARAMETER BLOCK FORMAT



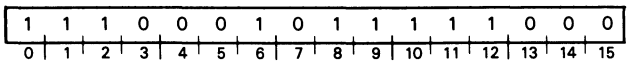
STATUS FLAGS UPDATED

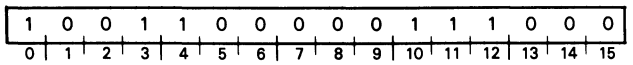


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

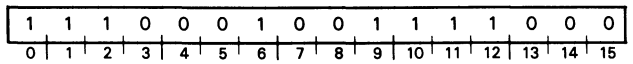
Evaluate Polynomial for Complex Array

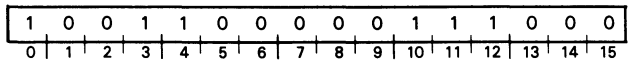
EPC





EPCP





$$W \leftarrow \sum_{n=0}^{N-1} X_n W^n$$

and, if:

Address (M)≠ 0

then

M ←W

ACTION - Places the complex scalar from the parameter block in the working register.* Then the instruction computes the value of the polynomial whose terms are stored in the source array X, and places the result in the working register. If parameter word 6 is nonzero, the result is also stored at the specified address, which may be anywhere in main memory.

> **NOTE:** *The working register is used as both input and output for this instruction. The result is not stored in the working register until the final value of the polynomial has been computed.*

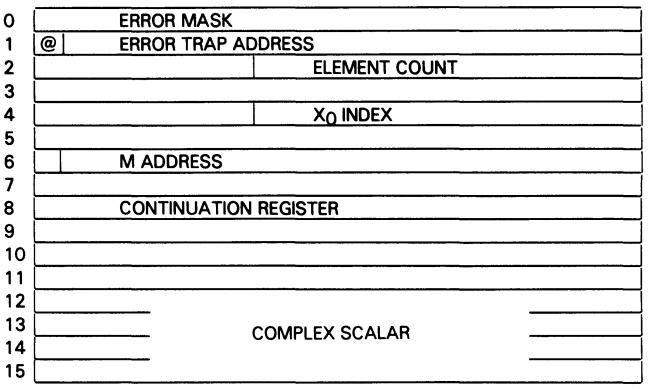
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

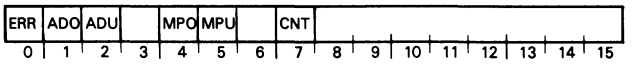
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of complex array X.
6	M ADDRESS	Address at which to store result.
8 *	---	Continuation register.
12-15 *	W	Complex scalar to be placed in working register.

*EPC only.

PARAMETER BLOCK FORMAT



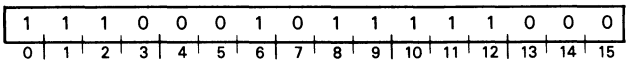
STATUS FLAGS UPDATED

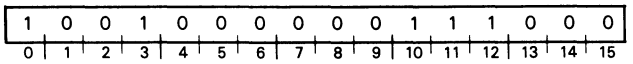


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

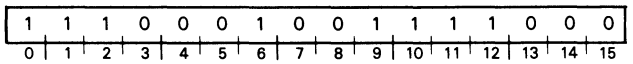
Evaluate Polynomial for Real Array

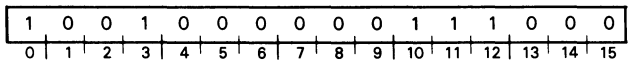
EPR





EPRP





$$W \leftarrow \sum_{n=0}^{N-1} X_n W^n$$

and, if:

Address (M)≠ 0

then

M ←W

ACTION - PLaces the real scalar from the parameter block in the working register.* Then the instruction computes the value of the polynomial whose coefficients are stored in the source array X, and places the result back in the working register. If parameter word 6 is nonzero, the result is also stored at the specified address, which may be anywhere in main memory.

NOTE: *The working register is used as both input and output by this instruction. The result is not stored in the working register until the final value of the polynomial has been computed.*

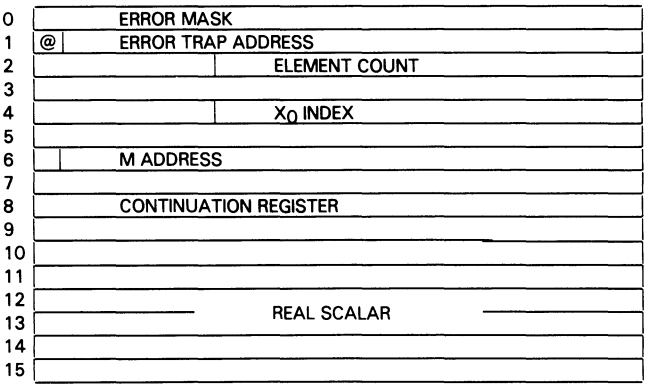
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

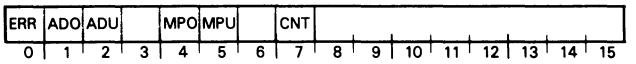
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
6	M ADDR	Address at which to store result.
8 *	---	Continuation register.
12,13 *	W	Real scalar to be placed in working register.

*EPR only.

PARAMETER BLOCK FORMAT



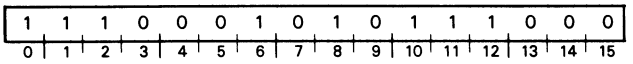
STATUS FLAGS UPDATED

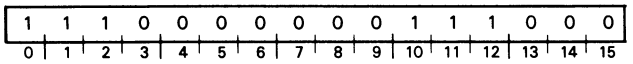


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

Fast Fourier Transform of Complex Array

FFTC





$$X_n \leftarrow \sum_{k=0}^{N-1} X_k \omega^{n-l}$$

$n = 0, 1, \ldots, N-1$

where

$\omega = e^{-2\pi i/N}$, for the forward transform,

$\omega = e^{2\pi i/N}$, for the inverse transform,

and

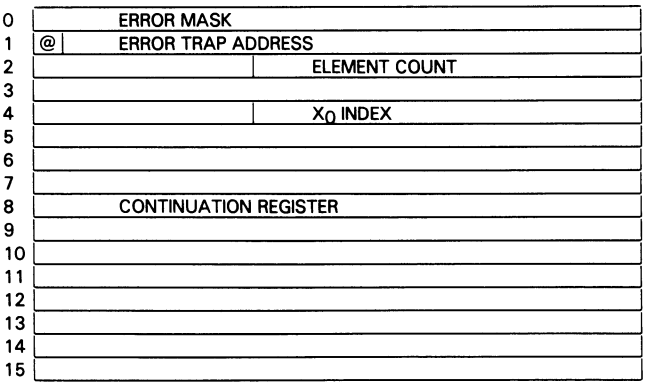
$i = \sqrt{-1}$

ACTION - Computes the discrete Fourier transform of the complex source array X. Either a forward or inverse transform may be calculated, depending on the value of bits 14 and 15 of the continuation register (parameter word 8). The result is returned to X in bit-reversed format. Thus the program will generally follow the **FFTC** with a bit-reversal instruction (**SCB** or **BRC**). The element count must be a power of 2. The format of the parameter block is as follows:

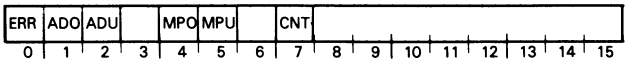
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count: must be a power of 2 in the range [8, 1024].
4	X_0 INDEX	Index of complex array X.
8	CONT REG	Continuation register: see table below.

PARAMETER BLOCK FORMAT

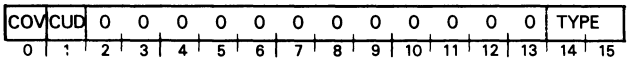


STATUS FLAGS UPDATED



BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

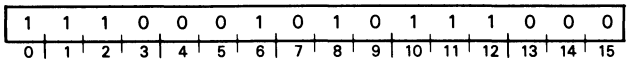
Continuation register:

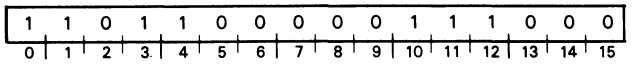


Bit	Name	Meaning when 1
0	COV	Continue on overflow.
1	CUD	Continue on underflow.
2-13		Reserved for future use.
14,15	FFT TYPE	Forward transform when bits = 01; Inverse transform when bits = 00.

Fast Fourier Transform of Real Array

FFTR





$$X_n \leftarrow 0.5\ [{}^*X_{N-n} + X_n + i\,\omega^{\,n}\,({}^*X_{N-n} - X_n)]$$

$n = 0, 1, \ldots, N-1$

where

$\omega = e^{-2\pi i/2N}$, for the forward transform,

$\omega = e^{2\pi i/2N}$, for the inverse transform,

*X is the complex conjugate of X,

and

$i = \sqrt{-1}$

ACTION - Computes the discrete Fourier transform of the source data. For a forward transform, the source data must be the result of a **FFTC** instruction. For an inverse transform, the result data of **FFTR** must be processed by an **FFTC** instruction. The direction of the transform (forward or inverse) is determined by bits in the continuation register as described below.

This instruction takes advantage of a short cut in calculating the real FFT. 2N real elements are interpreted as N complex elements and used in a complex FFT. The element count in the parameter block must be N, the number of *complex* elements.

If a 2048 (real) point transform is to be performed, the program must supply a pointer (in parameter word 6) to a table of real scalars. The table may be anywhere in main memory. Each of the 511 scalars must have a value equal to

$$\text{Cos}\ (2\pi n/2048)$$

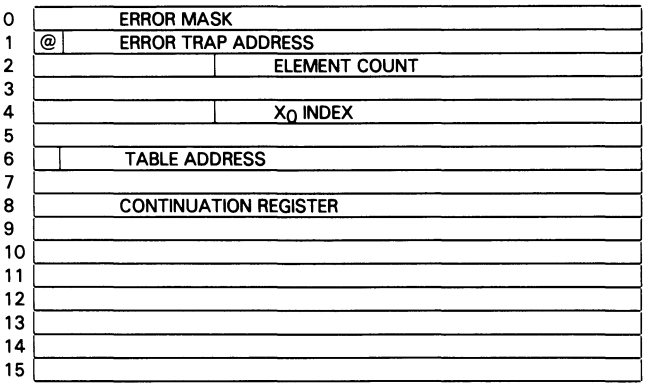
$n = 0, 1, \ldots, 511$

The format of the parameter block is as follows:

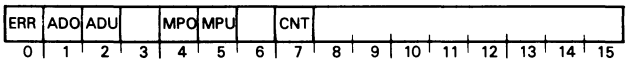
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count: must be a power of 2 in the range [8,1024].
4	X_0 INDEX	Index of real array X.
6	TABLE ADDR	Address of cosine table (if N = 1024).
8	CONT REG	Continuation register: see table below.

PARAMETER BLOCK FORMAT

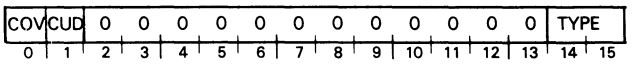


STATUS FLAGS UPDATED



BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

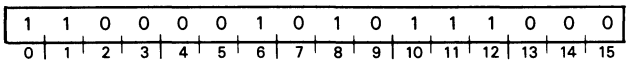
Continuation register:

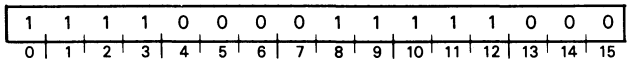


Bit	Name	Meaning when 1
0	COV	Continue on overflow.
1	CUD	Continue on underflow.
2-13		Reserved for future use.
14,15	FFT TYPE	For forward transform, bits = 01; For inverse transform, bits = 11.

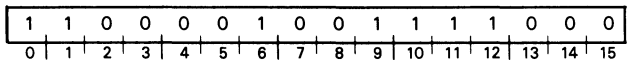
Float and Load (Convert Integer to Real)

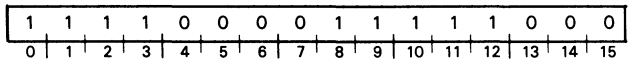
FLL





FLLP





$Z_{(n\ S_Z)} \leftarrow M_{(n\ S_m)}$

$n = 0, 1, \ldots, N\text{-}1$

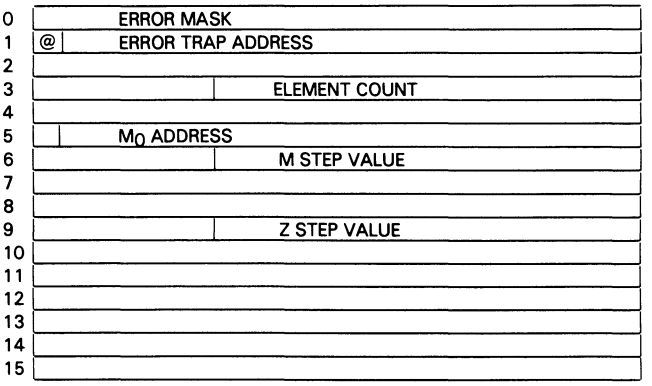
ACTION - Converts the integers in the source array M to ECLIPSE floating-point format, and places the results in the destination array Z. Note that Z will occupy twice as many words of memory as M. M may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

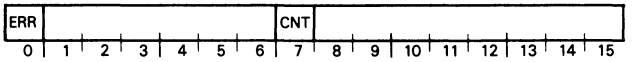
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	Z_0 INDEX	Index of real array Z.
6	M_0 ADDR	Main memory address of integer array M.
7	S_M	Step value for array M.
10 *	S_Z	Step value for array Z.

*FLL only.

PARAMETER BLOCK FORMAT



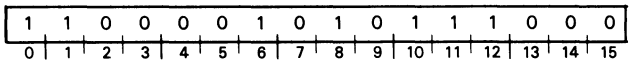
STATUS FLAGS UPDATED

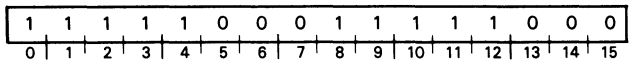


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

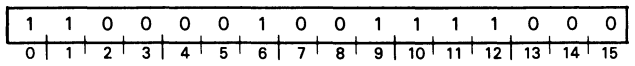
Fix and Store (Convert Real to Integer)

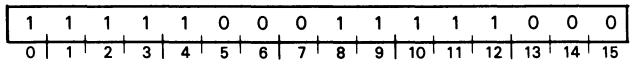
FXS





FXSP





$M_{(n\ S_m)} \leftarrow X_{(n\ S_x)}$

$n = 0, 1, \ldots, N\text{-}1$

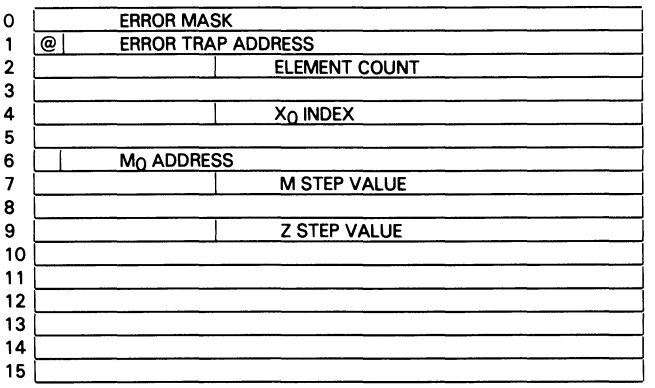
ACTION - Converts the elements of the real source array X to integers, and stores them in the destination array M. Note that M may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

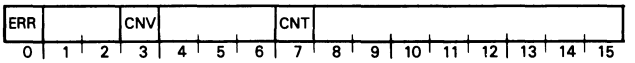
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	X_0 INDEX	Index of real array X.
6	M_0 ADDR	Main memory address of integer array M.
7	S_M	Step value for array M.
9 *	S_X	Step value for array X.

*FXS only.

PARAMETER BLOCK FORMAT



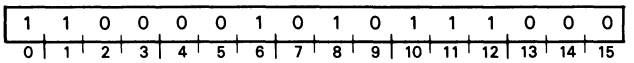
STATUS FLAGS UPDATED

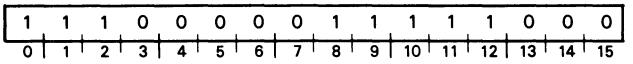


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
3	CNV	Conversion error.
7	CNT	Illegal element count.

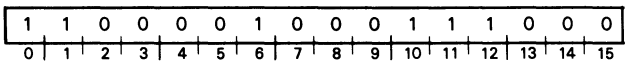
Integrate Real Array

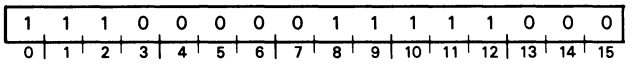
INR





INRP





$$Z_{(n\ S_Z)} \leftarrow \sum_{k=0}^{n} X_k\ S_X$$

n = 0, 1, . . . , N-1

ACTION - Computes the integral of the source array X and places the result in destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

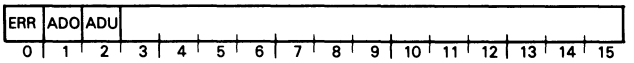
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.

*INR only.

PARAMETER BLOCK FORMAT

0	ERROR MASK		
1	@	ERROR TRAP ADDRESS	
2			ELEMENT COUNT
3			Z_0 INDEX
4			X_0 INDEX
5			
6			
7			
8	CONTINUATION REGISTER		
9			Z STEP VALUE
10			X STEP VALUE
11			
12			
13			
14			
15			

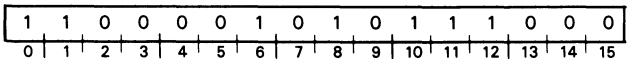
STATUS FLAGS UPDATED

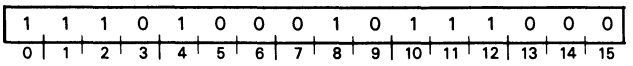


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.

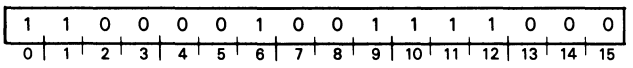
Inner Product of Complex Arrays

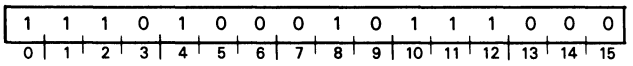
IPC





IPCP





$$W \leftarrow \sum_{n=0}^{N-1} X_{(n\,S_X)}\, Y_{(n\,S_Y)}$$

and, if:

Address (M) $\neq$ 0

then

M $\leftarrow$ W

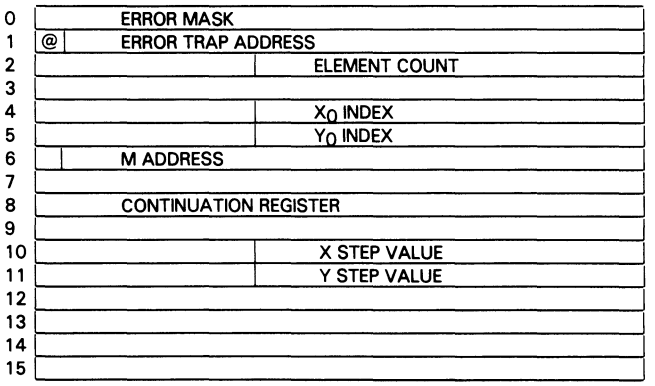
ACTION - Computes the inner ("dot") product of the two source arrays X and Y, and places the result in the working register. If parameter word 6 is nonzero, the result is also stored at the specified address, which may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

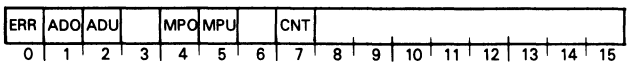
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of complex array X.
5	Y_0 INDEX	Index of complex array Y.
6	M ADDR	Address at which to store result.
8 *	---	Continuation register.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

*IPC only.

PARAMETER BLOCK FORMAT



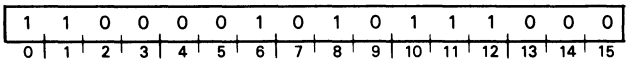
STATUS FLAGS UPDATED

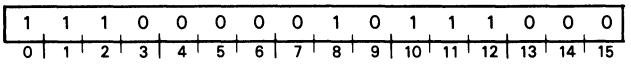


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

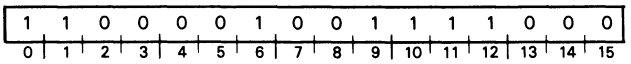
Inner Product of Real Arrays

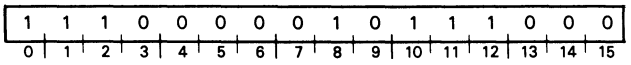
IPR





IPRP





$$W \leftarrow \sum_{n=0}^{N-1} X_{(n\,S_x)}\, Y_{(n\,S_y)}$$

and, if:

Address (M) $\neq$ 0

then

M $\leftarrow$ W

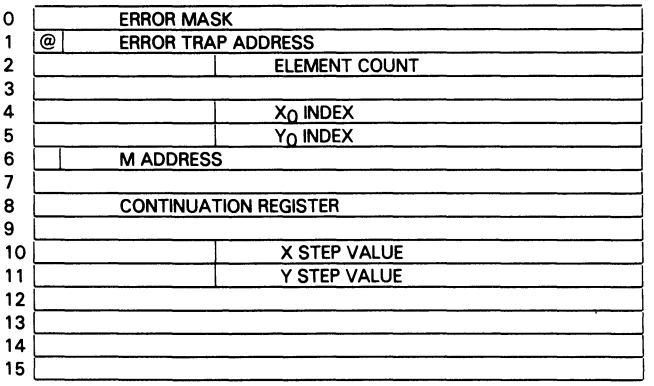
ACTION - Computes the inner ("dot") product of the two arrays X and Y, and places the result in the working register. If parameter word 6 is nonzero, the result is also stored at the specified address, which may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

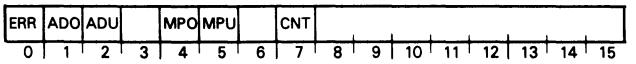
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
6	M ADDR	Address at which to store result.
8 *	---	Continuation register.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

IPR only.

PARAMETER BLOCK FORMAT



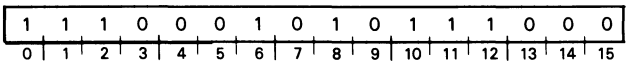
STATUS FLAGS UPDATED

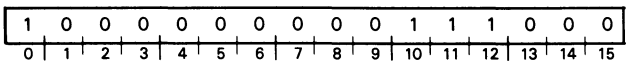


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

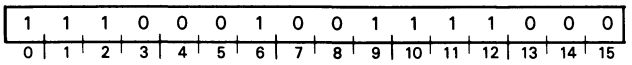
Load Complex Array

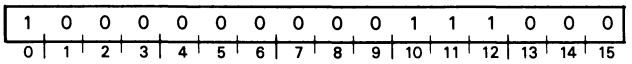
LDC





LDCP





$Z_{(n\ S_Z)} \leftarrow M_{(n\ S_M)}$

$n = 0, 1, \ldots, N\text{-}1$

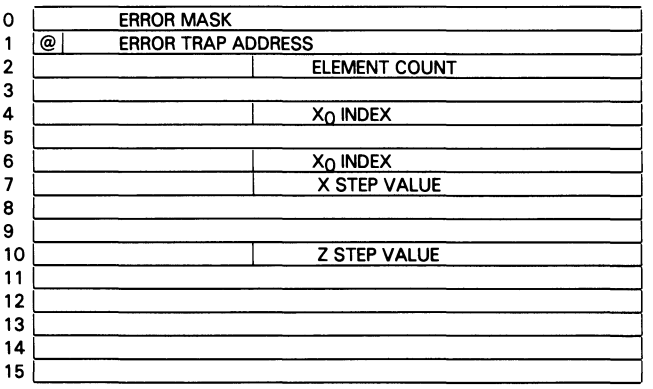
ACTION - Copies the source array M to the destination array Z. M may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

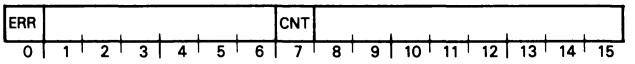
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	Z_0 INDEX	Index of complex array Z.
6	M_0 ADDR	Main memory address of complex array M.
7	S_M	Step value for array M.
10 *	S_Z	Step value for array Z.

*LDC only.

PARAMETER BLOCK FORMAT



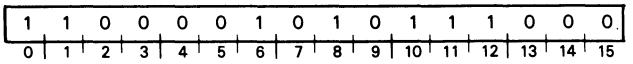
STATUS FLAGS UPDATED

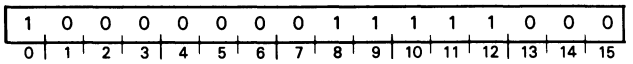


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

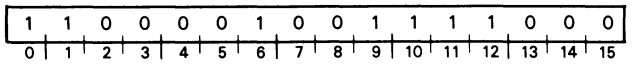
Load Real Array

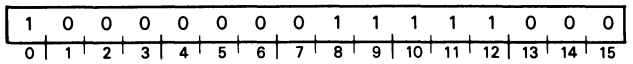
LDR





LDRP





$Z_{(n\ S_Z)} \leftarrow M_{(n\ S_m)}$

n = 0, 1, . . . , N-1

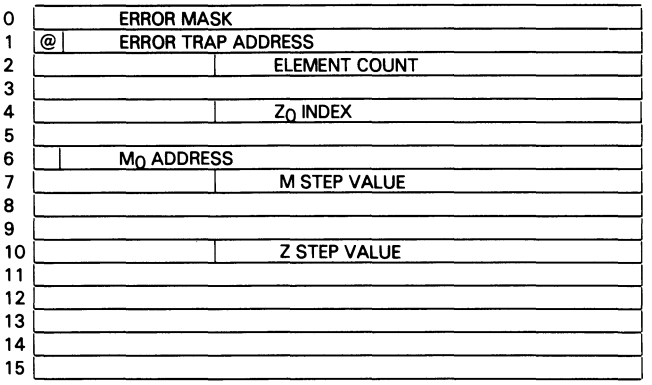
ACTION - Copies the source array X to the destination array Z. X may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

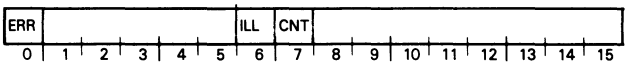
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	Z_0 INDEX	Index of real array Z.
6	M_0 ADDR	Main memory address of real array Z.
7	S_M	Step value for array M.
10 *	S_Z	Step value for array Z.

*LDR only.

PARAMETER BLOCK FORMAT



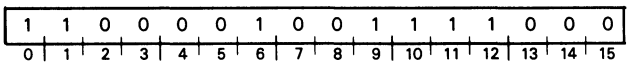
STATUS FLAGS UPDATED

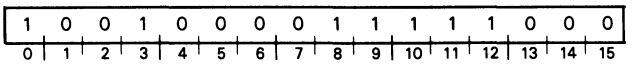


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
6	ILL	Undefined AP instruction.
7	CNT	Illegal element count.

Load Scratchpad Registers

LSR





$P_{i+n} \leftarrow M_{(n\,S_m)}$

$n = 0, 1, \ldots, N\text{-}1$

ACTION - Loads the specified group of scratchpad registers from the source array M. M may be anywhere in main memory. Note that two words of memory are required for each scratchpad register.

NOTES:

This instruction destroys the contents of the working register.

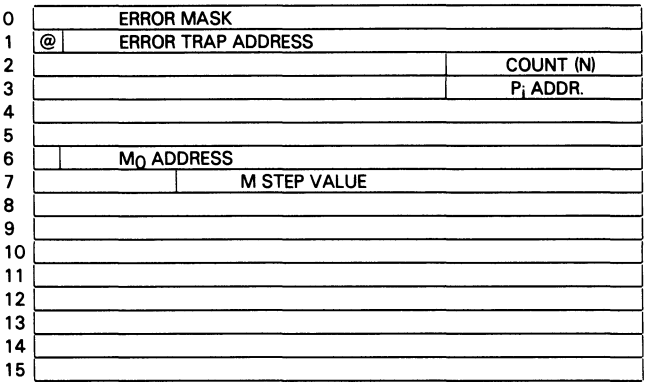
Locations 34_8 - 37_8 of scratchpad are reserved by hardware for use during interrupts.

The format of the parameter block is as follows:

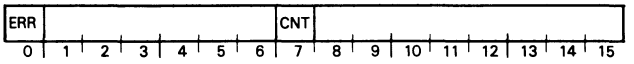
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	P_i ADDR	Scratchpad address of first element to load.
6	M_0 ADDR	Main memory address of integer array M.
7	S_M	Step value for array M.

PARAMETER BLOCK FORMAT



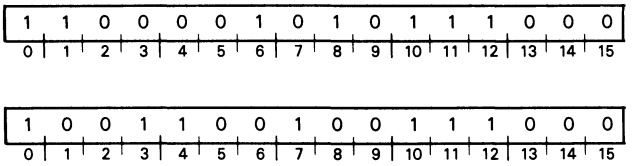
STATUS FLAGS UPDATED



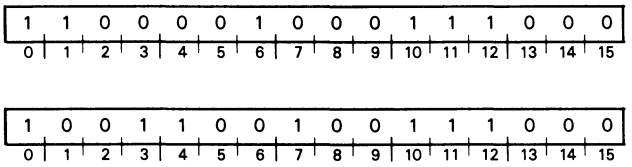
BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

Multiply Complex Arrays

MCA



MCAP



$Z_{(n\,S_Z)} \leftarrow X_{(n\,S_X)}\, Y_{(n\,S_Y)}$

$n = 0, 1, \ldots, N\text{-}1$

ACTION - Multiplies each element of the source array X by the corresponding element of the source array Y, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

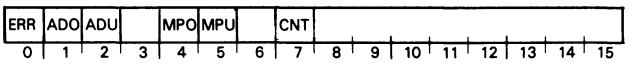
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of complex array Z.
4	X_0 INDEX	Index of complex array X.
5	Y_0 INDEX	Index of complex array Y.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

*MCA only.

PARAMETER BLOCK FORMAT

0	ERROR MASK	
1	@	ERROR TRAP ADDRESS
2		Z_0 INDEX
3		X_0 INDEX
4		Y_0 INDEX
5		
6		
7	CONTINUATION REGISTER	
8		Z STEP VALUE
9		X STEP VALUE
10		Y STEP VALUE
11		
12		
13		
14		
15		

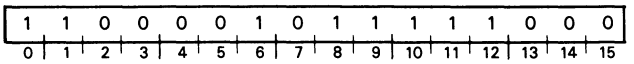
STATUS FLAGS UPDATED

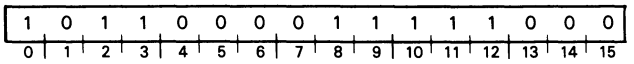


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

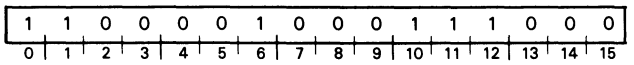
Multiply Complex Scalar by Array

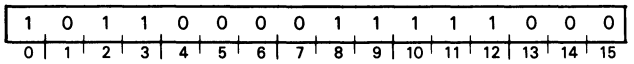
MCS





MCSP





$Z_{(n\ S_Z)} \leftarrow W\ X_{(n\ S_X)}$

$n = 0, 1, \ldots, N-1$

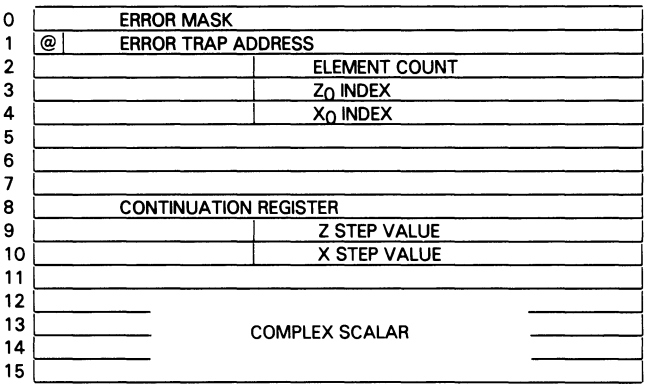
ACTION - Places the complex scalar in the parameter block in the working register.* Then the instruction multiplies the contents of the working register by each element of the complex source array X, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

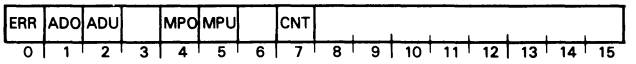
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of complex array Z.
4	X_0 INDEX	Index of complex array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
12-15 *	W	Complex scalar to be placed in working register.

*MCS only.

PARAMETER BLOCK FORMAT



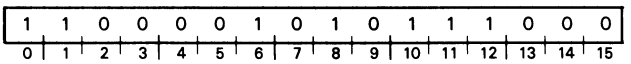
STATUS FLAGS UPDATED

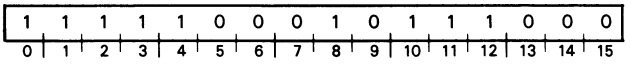


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

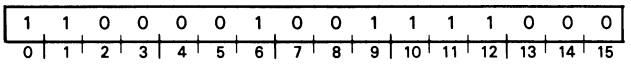
Minimum Element of Real Array

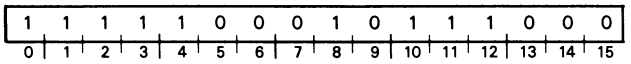
MNR





MNRP





$W \leftarrow Min\ (X_{(n\ S_X)})$

$n = 0, 1, \ldots, N-1$

and

$M_{Min} \leftarrow X_{(n\ S_X)}$,

and

$M_{index} \leftarrow n\ S_X$

for the first minimum element.

ACTION - Searches the source array X for the element having the lowest value, and places this value in the working register. Then the instruction stores a number indicating the location of this element in main memory at the address specified by parameter word 7. The number is relative to the start of X, and is multiplied by the current value of the X step register. The value of the minimum element is placed in main memory at the address specified by parameter word 6.

If several elements of X are equal to the minimum value, the address returned by MNR will point to the first occurence of the value.

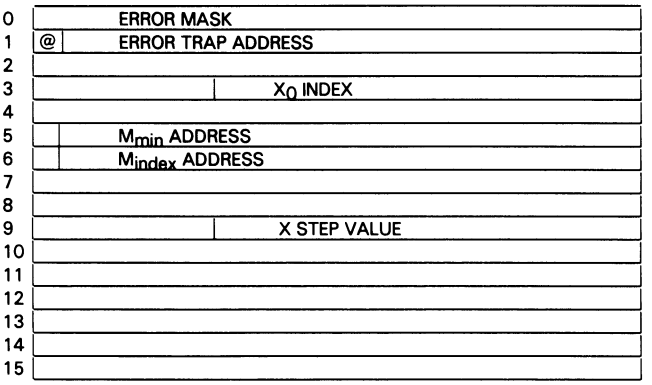
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

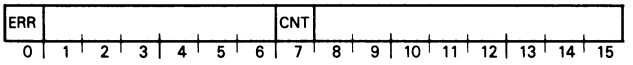
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
6	M_{min} ADDR	Address at which to store value of least element.
7	M_{index} ADDR	Address at which to store index of least element.
10 *	S_X	Step value for array X.

*MNR only.

PARAMETER BLOCK FORMAT



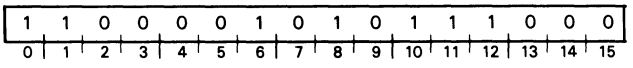
STATUS FLAGS UPDATED

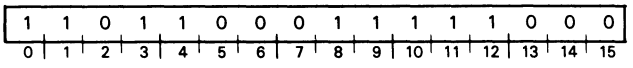


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

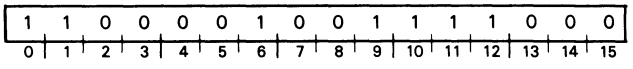
Modify Real Array

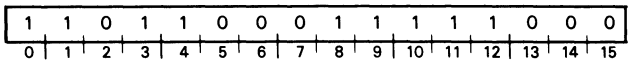
MOD





MODP





$Z_{(M_n)} \leftarrow X_{(n-1)\,S_X}$

$n = 1, 2, \ldots, N$

where

$N = M_0$

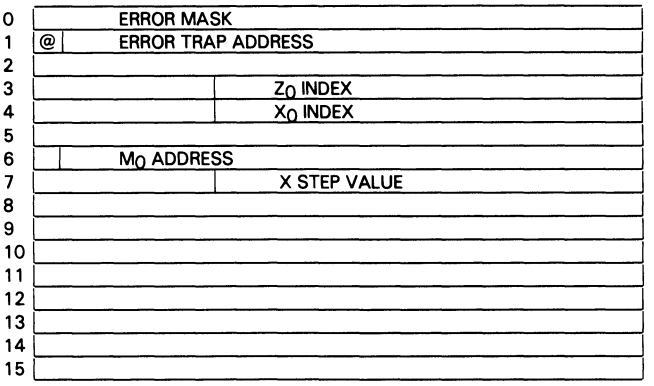
ACTION - Modifies the real array Z by taking elements of the source array X and placing them in Z at locations selected by elements of the integer array M. The 0th word of M contains its length. M may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

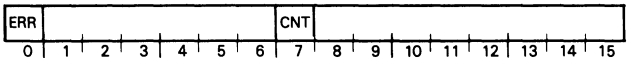
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
6	M_0 ADDR	Main memory address of integer array M.
10 *	S_X	Step value for array X.

*MOD only.

PARAMETER BLOCK FORMAT



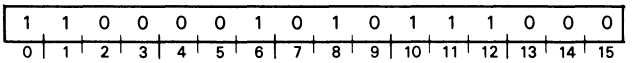
STATUS FLAGS UPDATED

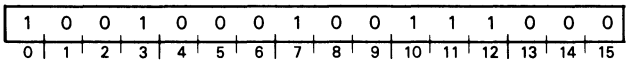


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

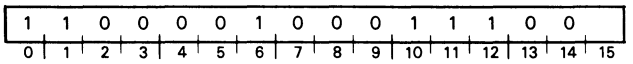
Multiply Real Arrays

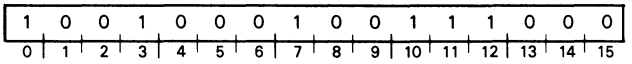
MRA





MRAP





$Z_{(n\ S_Z)} \leftarrow X_{(n\ S_X)}\ Y_{(n\ S_Y)}$

$n = 0, 1, \ldots, N\text{-}1$

ACTION - Multiplies each element of the source array X by the corresponding element of the source array Y, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

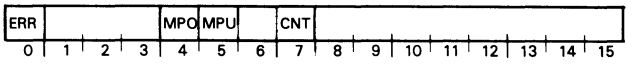
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

*MRA only.

PARAMETER BLOCK FORMAT

0	ERROR MASK	
1	@	ERROR TRAP ADDRESS
2		ELEMENT COUNT
3		Z_0 INDEX
4		X_0 INDEX
5		Y_0 INDEX
6		
7		
8	CONTINUATION REGISTER	
9		Z STEP VALUE
10		X STEP VALUE
11		Y STEP VALUE
12		
13		
14		
15		

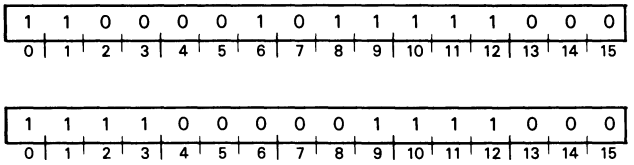
STATUS FLAGS UPDATED



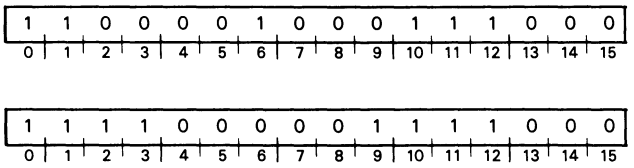
BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

Multiply Real Scalar by Array

MRS



MRSP



$Z_{(n\ S_Z)} \leftarrow W\ X_{(n\ S_X)}$

$n = 0, 1, \ldots, N-1$

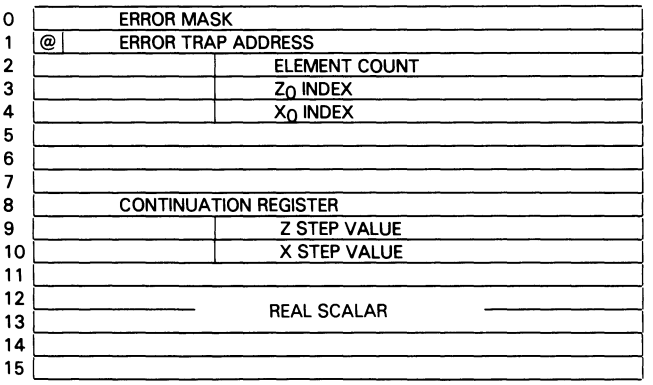
ACTION - Places the real scalar from the parameter block in the working register.* Then the instruction multiplies the real scalar in the working register by each element of the source array X, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

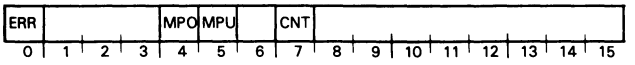
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
12,13 *	W	Real scalar to be placed in working register.

*MRS only.

PARAMETER BLOCK FORMAT



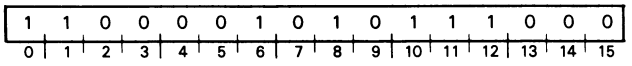
STATUS FLAGS UPDATED

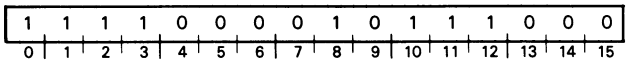


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

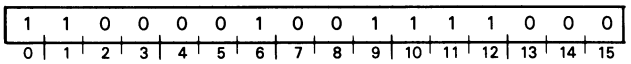
Maximum Element of Real Array

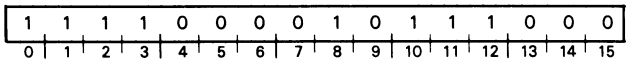
MXR





MXRP





$W \leftarrow Max\ (X_{(n\ S_X)})$

$n = 0, 1, \ldots, N\text{-}1$

and

$M_{Max} \leftarrow X_{(n\ S_X)}$,

and

$M_{index} \leftarrow n\ S_x$

for the first maximum element.

ACTION - Searches the source array X for the element having the greatest value, and places this value in the working register. Then the instruction stores a number indicating the location of this element In main memory at the address specified by parameter word 7. The number is relative to the start of X, and is multiplied by the current value of the X step register. The value of the maximum element is placed in main memory at the address specified by parameter word 6.

If several elements of X are equal to the maximum value, the address returned by **MXR** will point to the first occurence of the value.

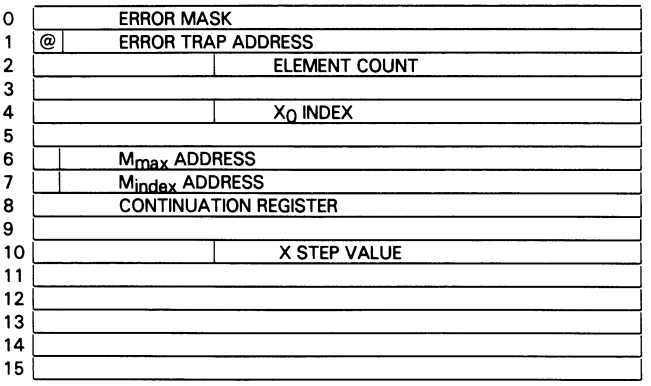
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

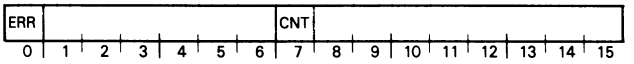
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
6	M_{max} ADDR	Address at which to store value of greatest element.
7	M_{index} ADDR	Address at which to store index of greatest element.
8 *	---	Continuation register.
10 *	S_X	Step value for array X.

*MXR only.

PARAMETER BLOCK FORMAT



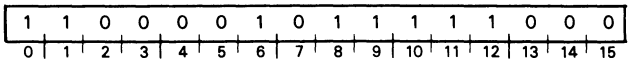
STATUS FLAGS UPDATED

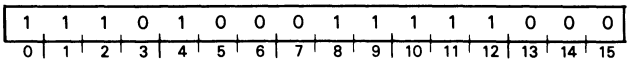


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

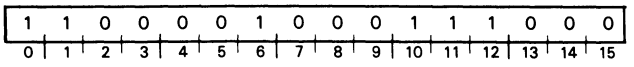
Negate Real Array

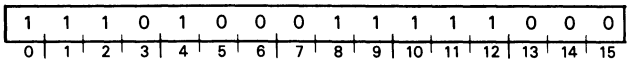
NRA





NRAP





$Z_{(n\ S_Z)} \leftarrow W - X_{(n\ S_X)}$

n = 0, 1, . . . , N-1

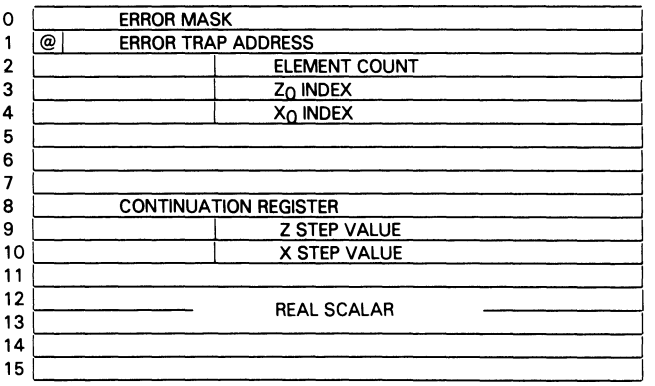
ACTION - Places the real scalar from the parameter block in the working register.* Then the instruction subtracts each element of the real source array X from the contents of the working register, and places the results in the destination array Z. (The contents of the working register are not changed by the subtractions.) The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

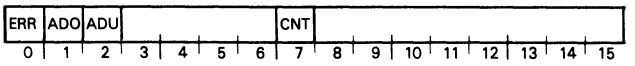
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
12,13 *	W	Real scalar to be placed in working register.

*NRA only.

PARAMETER BLOCK FORMAT



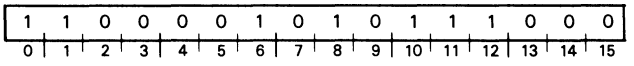
STATUS FLAGS UPDATED

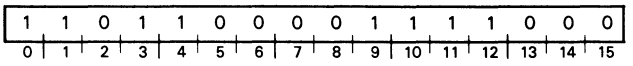


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
7	CNT	Illegal element count.

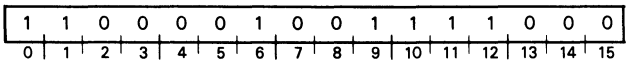
Product of Elements of Complex Array

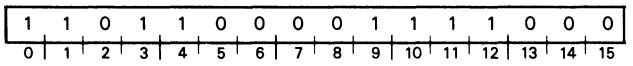
PEC





PECP





$$W \leftarrow \prod_{n=0}^{N-1} X_{(n\,S_X)}$$

and, if:

Address (M)$\neq$ 0

then

M $\leftarrow$W

ACTION - Multiplies the elements of the source array X and places the result in the working register. If parameter word 6 is nonzero, the result is also stored at the specified address, which may be anywhere in main memory.

NOTE: *The previous contents of the working register are destroyed by this instruction.*

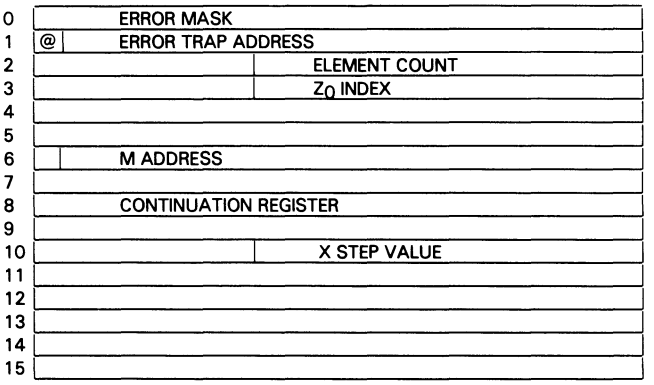
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

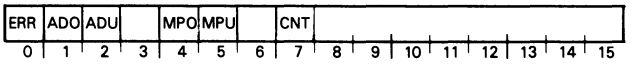
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of complex array X.
6	M ADDRESS	Address at which to store result.
8 *	---	Continuation register.
10 *	S_X	Step value for array X.

*PEC only.

PARAMETER BLOCK FORMAT



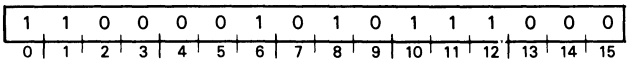
STATUS FLAGS UPDATED

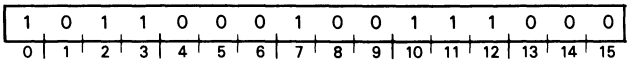


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

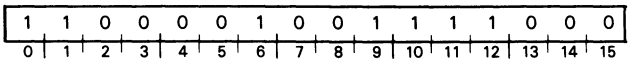
Product of Elements of Real Array

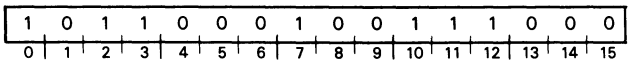
PER





PERP





$$W \leftarrow \prod_{n=0}^{N-1} X_{(n\,S_X)},$$

and, if:

Address (M) $\neq$ 0

then

M $\leftarrow$ W

ACTION - Multiplies the elements of the source array X and places the result in the working register. If parameter word 6 is nonzero, the result is also stored at the specified address, which may be anywhere in main memory.

NOTE: *The previous contents of the working register are destroyed by this instruction.*

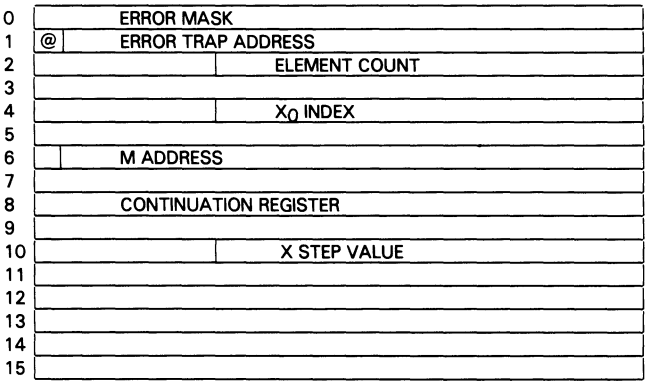
The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

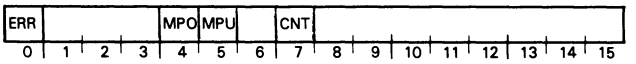
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
6	M ADDR	Address at which to store result.
8 *	---	Continuation register.
10 *	S_X	Step value for array X.

*PER only.

PARAMETER BLOCK FORMAT



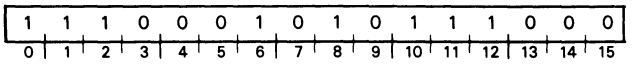
STATUS FLAGS UPDATED

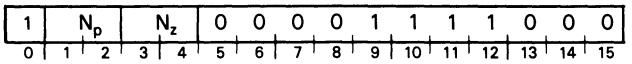


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

Recursive Filter for Complex Data

RFC N_p, N_z





$Z_j \leftarrow Q_{jD}$

For all j such that

$0 \leq jD \leq N-1$

given

$A_0 = 1$,

and

$$Q_n = \sum_{k=0}^{Np-1} A_k X_{n-k} - \sum_{l=1}^{Nz-1} B_l Q_{n-l}$$

$n = 0, 1, \ldots, N-1$

ACTION - Performs a filter function of the type specified by N_p and N_z on the input data in the source array X. N_p is the number of poles of the filter, which must be 1 or 2. N_z is the number of zeroes of the filter, which must be 0, 1, or 2. X must have $N + N_z$ elements.

The output is placed in the destination array Z. The elements Z_{-1} and Z_{-2} must be available; thus Z actually contains N + 2 elements. Z_{-1} and Z_{-2} may or may not be used as initial conditions, depending on the specified values of N_p and N_z. However they must be allocated since they are used by the AP for temporary storage.

Several elements of AP scratchpad are used to store coefficients. Each coefficient takes two elements of scratchpad, since they are complex numbers. The use of scratchpad is summarized in the table below.

FILTER TYPE		SCRATCHPAD ADDRESS			
N_p	N_z	0,1	2,3	4,5	6,7
1	0	-B1	unused	unused	unused
	1	A1	-B1	unused	unused
	2	A1	-B1	A2	unused
2	0	-B2	-B1	unused	unused
	1	-B2	-B1	A1	unused
	2	-B2	-B1	A1	A2

The equation above shows a temporary array Q. This array, included in the equation for clarity, has no physical representation in memory and is completely transparent to the user.

The desampling rate D must be specified in the parameter block. D is a positive integer less than or equal to N.

NOTES: *The sizes of the arrays are limited by the condition that* $2N + N_z + N_p$ *be less than or equal to 1024.*

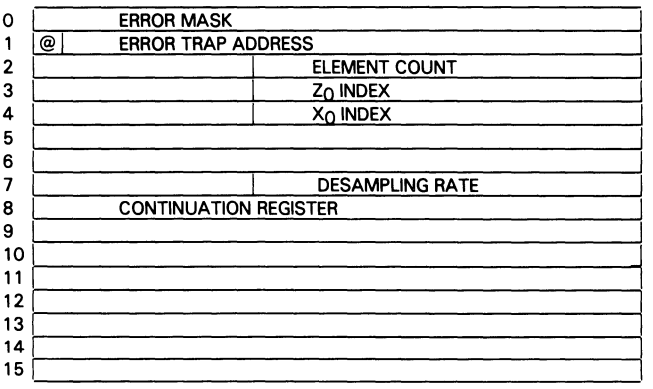
RFC *destroys the contents of the step registers and the working register.*

The format of the parameter block is as follows:

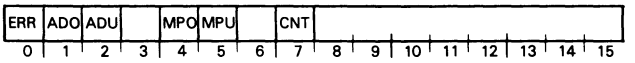
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count N used to compute the length of source array X.
3	Z_0 INDEX	Index of complex array Z.
4	X_0 INDEX	Index of complex array X.
7	D	Desampling rate.
8	---	Continuation register.

PARAMETER BLOCK FORMAT



STATUS FLAGS UPDATED

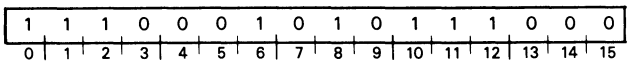


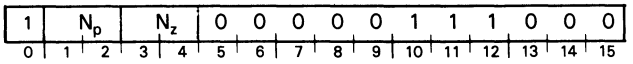
BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

This page intentionally left blank.

Recursive Filter for Real Data

RFR N_p , N_z





$Z_j \leftarrow Q_{jD}$

For all j such that

$0 \leq jD \leq N\text{-}1$

given

$A_0 = 1$,

and

$$Q_n = \sum_{k=0}^{Np-1} A_k X_{n-k} - \sum_{l=1}^{Nz-1} B_l Q_{n-l}$$

$n = 0, 1, \ldots, N-1$

ACTION - Performs a filter function of the type specified by N_p and N_z on the input data in the source array X. N_p is the number of poles of the filter, which must be 1 or 2. N_z is the number of zeroes of the filter, which must be 0, 1, or 2. X must have $N + N_z$ elements.

The output is placed in the destination array Z. The elements Z_{-1} and Z_{-2} must be available; thus Z actually contains N + 2 elements. Z_{-1} and Z_{-2} may or may not be used as initial conditions, depending on the specified values of N_p and N_z. However they must be allocated since they are used by the AP for temporary storage.

The equation above shows a temporary intermediate array Q. This array, included in the equation for clarity, has no physical representation in memory and is completely transparent to the user.

Several elements of AP scratchpad are used to store coefficients. The use of scratchpad is summarized in the table below.

FILTER TYPE		SCRATCHPAD ADDRESS			
N_p	N_z	0	1	2	3
1	0	-B1	0	0	0
1	1	-B1	0	A1	0
1	2	-B1	0	A1	A2
2	0	-B1	-B2	0	unused
2	1	-B1	-B2	A1	unused
2	2	-B1	-B2	A1	A2

The desampling rate D must be specified in the parameter block. D is a positive integer less than or equal to N.

NOTES: *The sizes of the arrays are limited by the condition that* $2N + N_z + N_p$ *be less than or equal to 2048.*

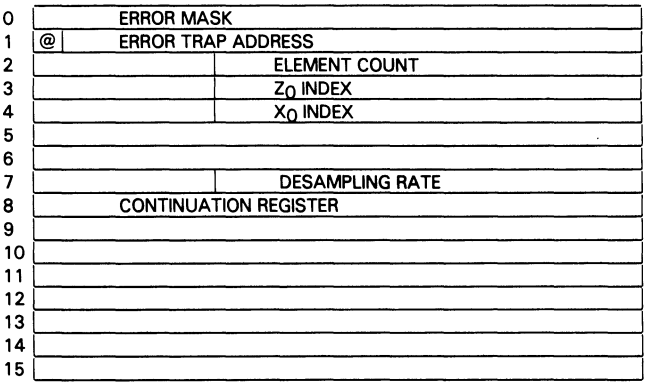
RFR *destroys the contents of the step registers and the working registers.*

The format of the parameter block is as follows:

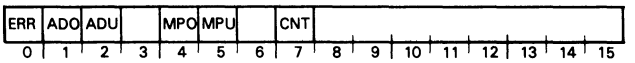
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count used to compute the length of source array X.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
7	D	Desampling rate.
8	---	Continuation register.

PARAMETER BLOCK FORMAT



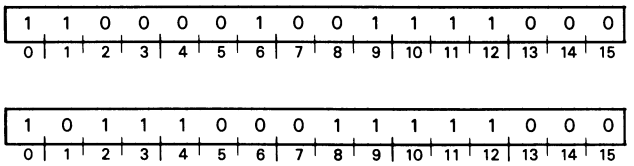
STATUS FLAGS UPDATED



BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

Store Complex Array Bit-reversed

SCB



$M_{(n\ S_m)} \leftarrow X_{n'}$,

n = 0, 1, . . . , N-1

where

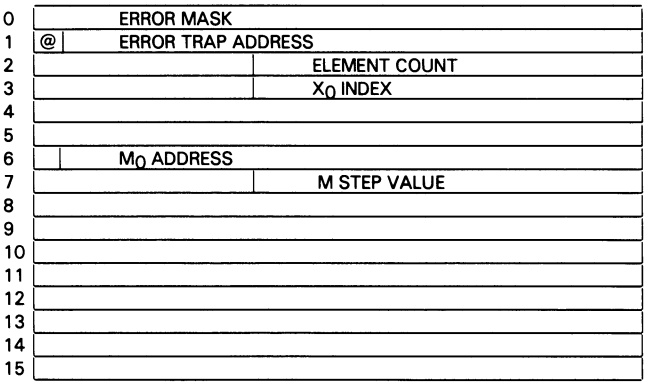
n' = Bit reverse of n within $\log_2 N$ bits.

ACTION - Copies the elements of the source array X to the destination array M with bit-reversed indexing. For example, with an element count of 16, X_1 would be moved to M_8, since 8 (1000_2) is the bit-reverse of 1 (0001_2. Similarly, X_2 would be moved to M_4. M may be anywhere in main memory. The format of the parameter block is as follows:)

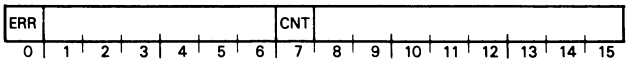
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count; must be a power of two less than or equal to 1024.
3	X_0 INDEX	Index of complex array X.
6	M_0 ADDR ADDR	Main memory address of complex array M.
7	S_M	Step value for array M.

PARAMETER BLOCK FORMAT



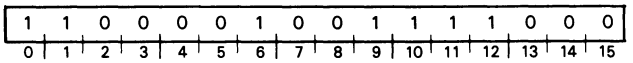
STATUS FLAGS UPDATED

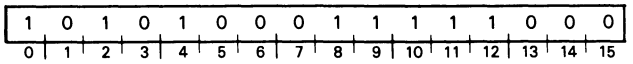


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

Store Complex Scalar from Working Register

SCW





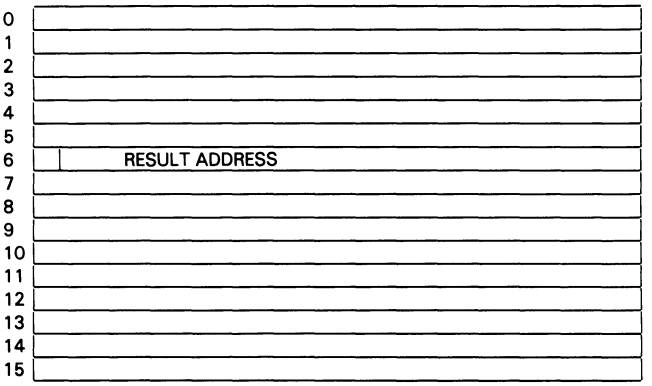
M ← W

ACTION - Places the complex scalar contained in the working register in four words of main memory starting at the specified address. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

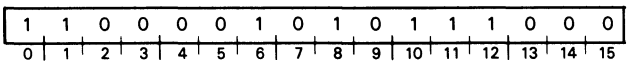
WORD	NAME	CONTENTS
6	W ADDR	Address at which to store scalar.

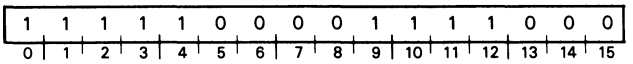
PARAMETER BLOCK FORMAT



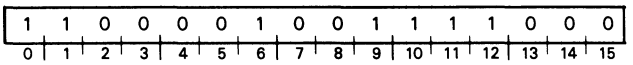
Sum of Elements of Real Array

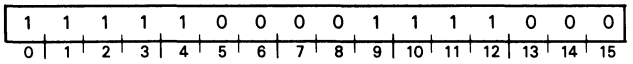
SER





SERP





$$W \leftarrow \sum_{n=0}^{N-1} X_{(n\,S_X)},$$

and, if:

Address (M) $\neq$ 0

then

M $\leftarrow$ W

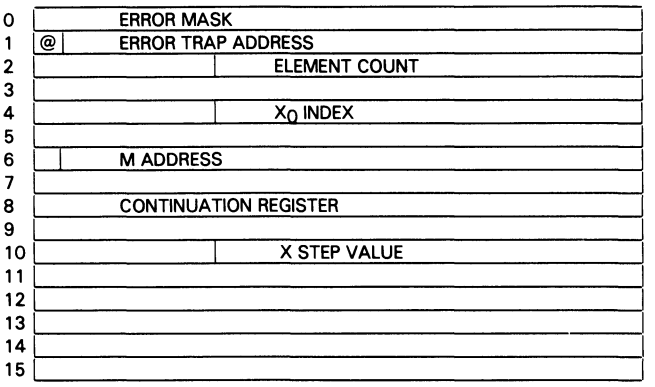
Adds the elements of the source array X together and places the result in the working register. If parameter word 6 is nonzero, the result is also stored at the specified address, which may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

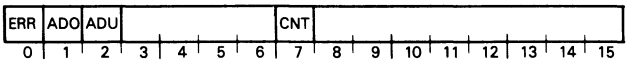
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
4	X_0 INDEX	Index of real array X.
6	M ADDR	Address at which to store result.
8 *	---	Continuation register.
10 *	S_X	Step value for array X.

*SER only.

PARAMETER BLOCK FORMAT



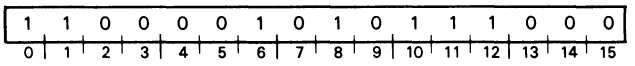
STATUS FLAGS UPDATED

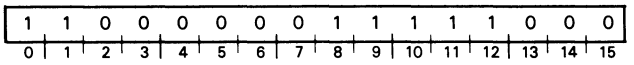


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
7	CNT	Illegal element count.

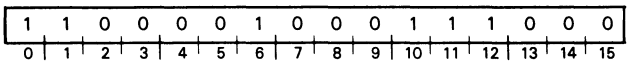
Square Magnitudes of Complex Arrays

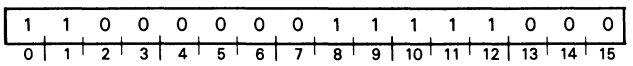
SMA





SMAP





$$Z_{(n\ S_Z)} \leftarrow [Re\ (X_{(n\ S_X)})]^2 + [Im\ (X_{(n\ S_X)})]^2$$

$n = 0, 1, \ldots, N\text{-}1$

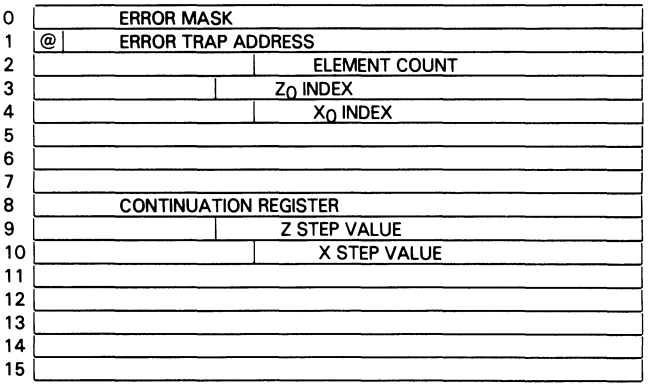
ACTION - Computes the square of the magnitude of each element of the source array X, and places the results in the destination array Z. Note that the source array is complex, but the destination array is real. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

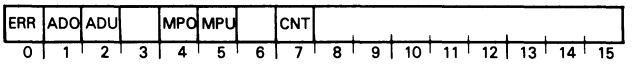
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Element address of real array Z.
4	X_0 INDEX	Index of complex array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.

*SMA only.

PARAMETER BLOCK FORMAT



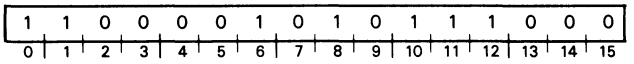
STATUS FLAGS UPDATED

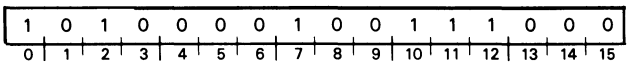


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

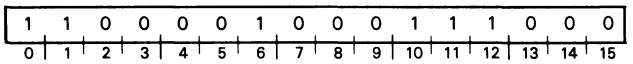
Signed Product of Real Arrays

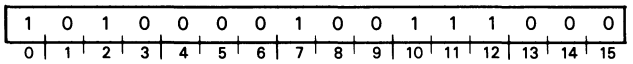
SPR





SPRP





$Z_{(n\ S_Z)} \leftarrow |X_{(n\ S_X)}| Y_{(n\ S_Y)}$

$n = 0, 1, \ldots, N-1$

ACTION - Multiplies the absolute value of each element of the source array X by the corresponding element of source array Y, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

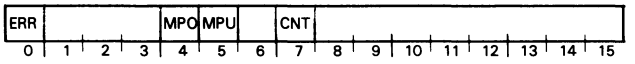
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

*SPR only.

PARAMETER BLOCK FORMAT

0	ERROR MASK	
1	@	ERROR TRAP ADDRESS
2		ELEMENT COUNT
3		Z_0 INDEX
4		X_0 INDEX
5		Y_0 INDEX
6		
7		
8	CONTINUATION REGISTER	
9		Z STEP VALUE
10		X STEP VALUE
11		Y STEP VALUE
12		
13		
14		
15		

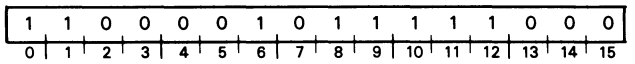
STATUS FLAGS UPDATED

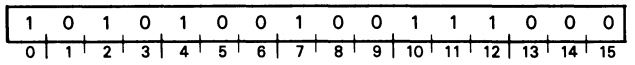


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

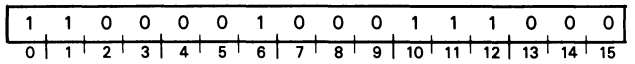
Signed Product of Real Scalar and Array

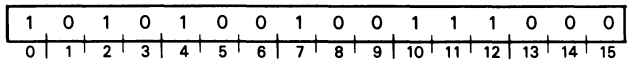
SPS





SPSP





$Z_{(n\ S_Z)} \leftarrow |X_{(n\ S_X)}| W$

$n = 0, 1, \ldots, N-1$

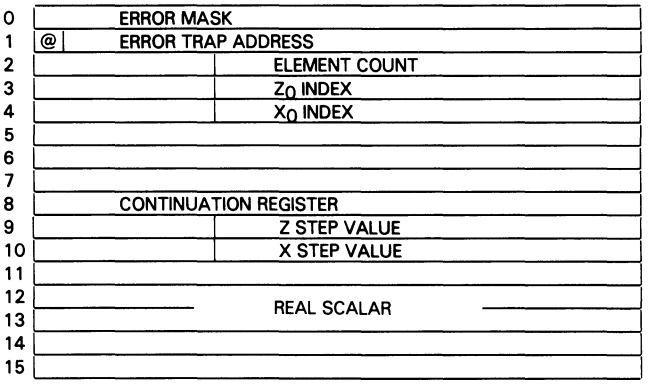
ACTION - Places the real scalar from the parameter block in the working register.* Then the instruction multiplies the absolute value of each element of the real source array X by the contents of the working register, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

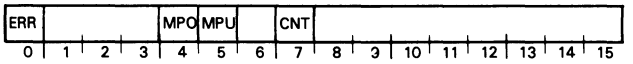
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
12,13 *	W	Real scalar to be placed in working register.

*SPS only.

PARAMETER BLOCK FORMAT



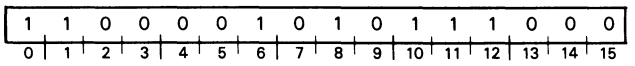
STATUS FLAGS UPDATED

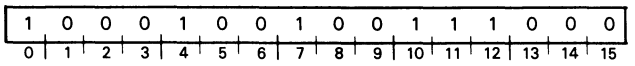


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
4	MPO	Multiplication result overflow.
5	MPU	Multiplication result underflow.
7	CNT	Illegal element count.

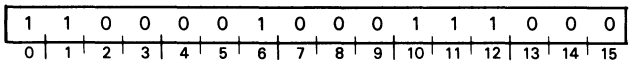
Subtract Real Arrays

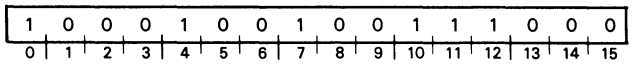
SRA





SRAP





$Z_{(n\ S_Z)} \leftarrow X_{(n\ S_X)} - Y_{(n\ S_Y)}$

$n = 0, 1, \ldots, N-1$

ACTION - Subtracts each element of the source array Y from the corresponding element of X, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

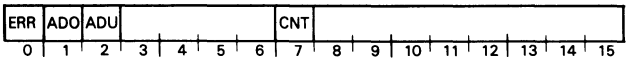
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
5	Y_0 INDEX	Index of real array Y.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
11 *	S_Y	Step value for array Y.

*SRA only.

PARAMETER BLOCK FORMAT

0	ERROR MASK	
1	@ ERROR TRAP ADDRESS	
2		ELEMENT COUNT
3		Z_0 INDEX
4		X_0 INDEX
5		Y_0 INDEX
6		
7		
8	CONTINUATION REGISTER	
9		Z STEP VALUE
10		X STEP VALUE
11		Y STEP VALUE
12		
13		
14		
15		

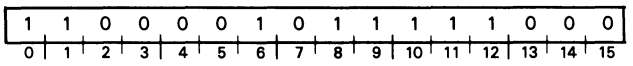
STATUS FLAGS UPDATED

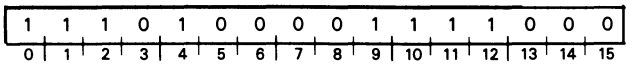


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
7	CNT	Illegal element count.

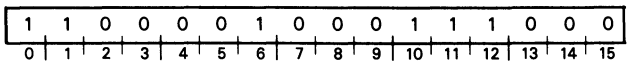
Subtract Real Scalar from Array

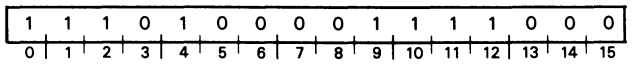
SRS





SRSP





$Z_{(n\ S_Z)} \leftarrow X_{(n\ S_X)} - W$

$n = 0, 1, \ldots, N-1$

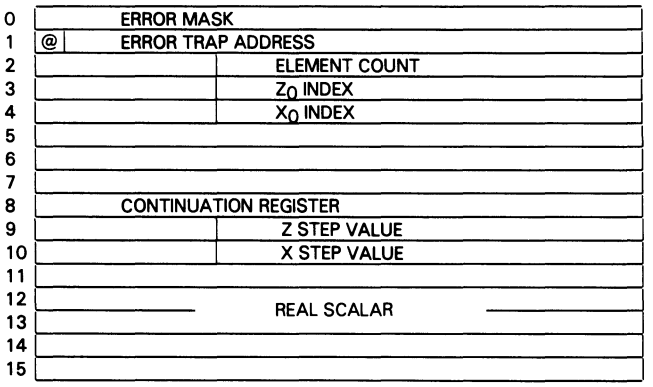
ACTION - Places the real scalar from the parameter block in the working register.* Then the instruction subtracts the contents of the working register from each element of the source array X, and places the results in the destination array Z. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

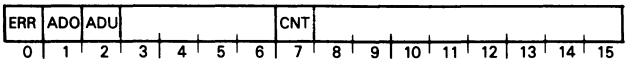
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	Z_0 INDEX	Index of real array Z.
4	X_0 INDEX	Index of real array X.
8 *	---	Continuation register.
9 *	S_Z	Step value for array Z.
10 *	S_X	Step value for array X.
12,13 *	W	Real scalar to be placed in working register.

*SRS only.

PARAMETER BLOCK FORMAT



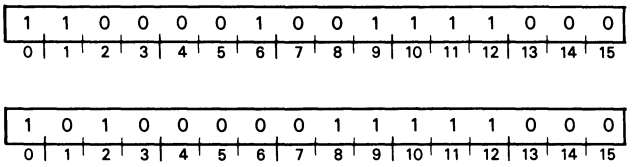
STATUS FLAGS UPDATED



BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
1	ADO	Addition result overflow.
2	ADU	Addition result underflow.
7	CNT	Illegal element count.

Store Real Scalar from Working Register

SRW



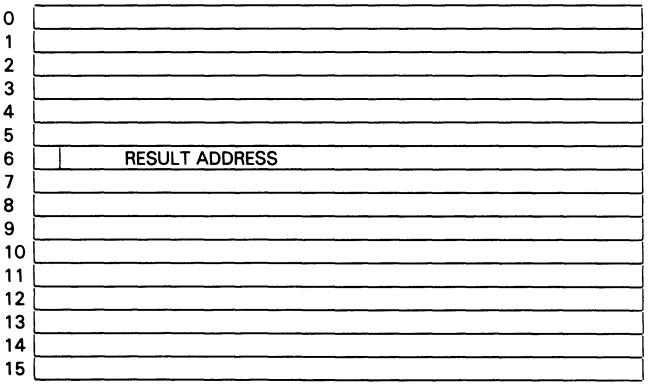
M ← W

ACTION - Places the real scalar contained in the working register in two words of main memory at the specified address. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

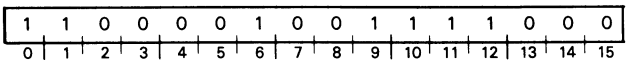
WORD	NAME	CONTENTS
6	W ADDR	Address at which to store scalar.

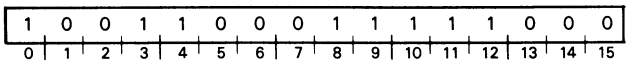
PARAMETER BLOCK FORMAT



Store Scratchpad Registers

SSR





$M_{(n\ S_m)} \leftarrow P_{i+n}$

$n = 0, 1, \ldots, N-1$

ACTION - Stores the specified group of scratchpad registers into the destination array M. M may be anywhere in main memory. Note that two words of memory are required for each scratchpad register.

NOTES:

This instruction destroys the contents of the working register.

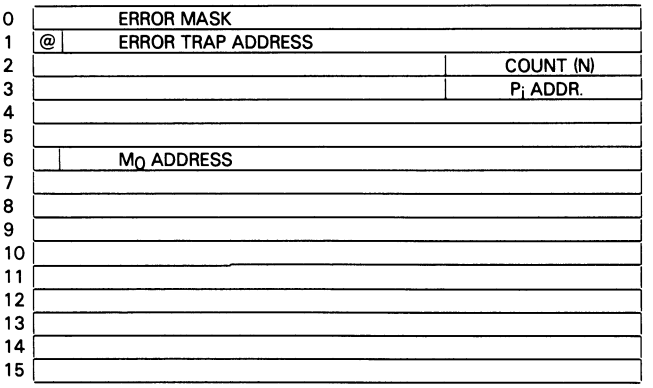
Locations 34_8 - 37_8 of scratchpad are reserved by hardware for use during interrupts.

The format of the parameter block is as follows:

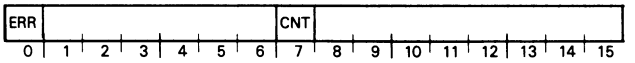
PARAMETER BLOCK CONTENTS

WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	P_i ADDR	Scratchpad address of first register to store.
6	M_0 ADDR	Main memory address of integer array M.

PARAMETER BLOCK FORMAT



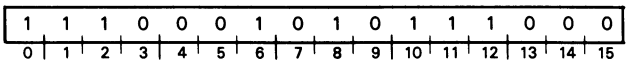
STATUS FLAGS UPDATED

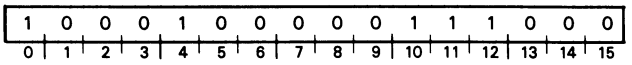


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

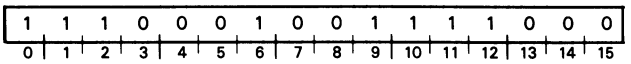
Store Complex Array

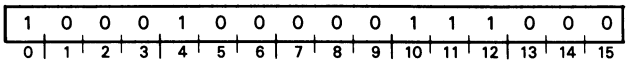
STC





STCP





$M_{(n\ S_m)} \leftarrow X_{(n\ S_x)}$

$n = 0, 1, \ldots, N-1$

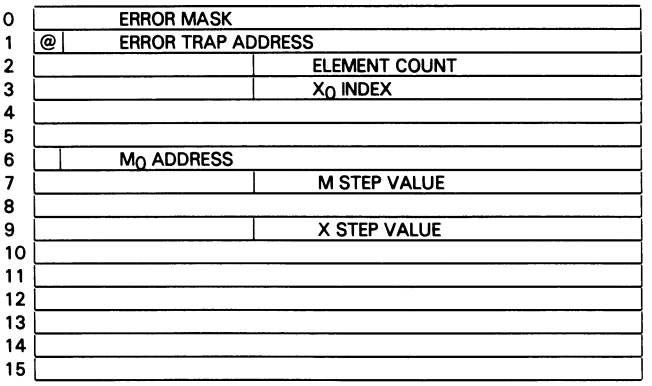
ACTION - Copies the source array X to the destination array M. M may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

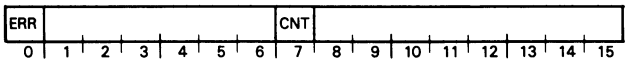
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	X_0 INDEX	Index of complex array X.
6	M_0 ADDR	Main memory address of complex array M.
7	S_M	Step value for array M.
9 *	S_X	Step value for array X.

*STC only.

PARAMETER BLOCK FORMAT



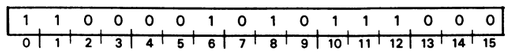
STATUS FLAGS UPDATED

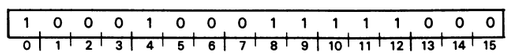


BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

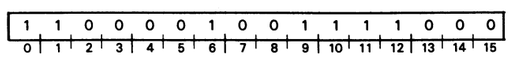
Store Real Array

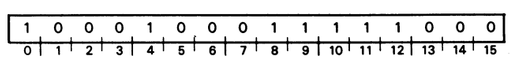
STR





STRP





$M_{(n\ S_m)} \leftarrow X_{(n\ S_X)}$

n = 0, 1, . . . , N-1

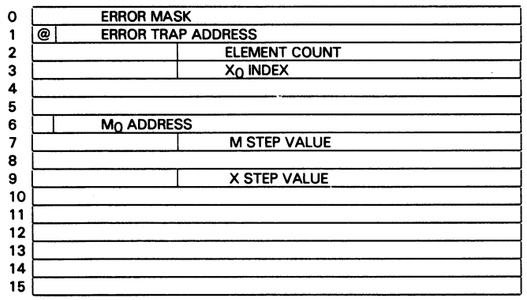
ACTION - Copies the source array X to the destination array M. M may be anywhere in main memory. The format of the parameter block is as follows:

PARAMETER BLOCK CONTENTS

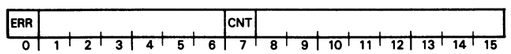
WORD	NAME	CONTENTS
0	---	Error mask.
1	---	Trap address.
2	N	Element count.
3	X_0 INDEX	Index of real array X.
6	M_0 ADDR	Main memory address of real array M.
7	S_M	Step value for array M.
9 *	S_X	Step value for array X.

*STR only.

PARAMETER BLOCK FORMAT



STATUS FLAGS UPDATED



BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least one of bits 1-7 is 1.
7	CNT	Illegal element count.

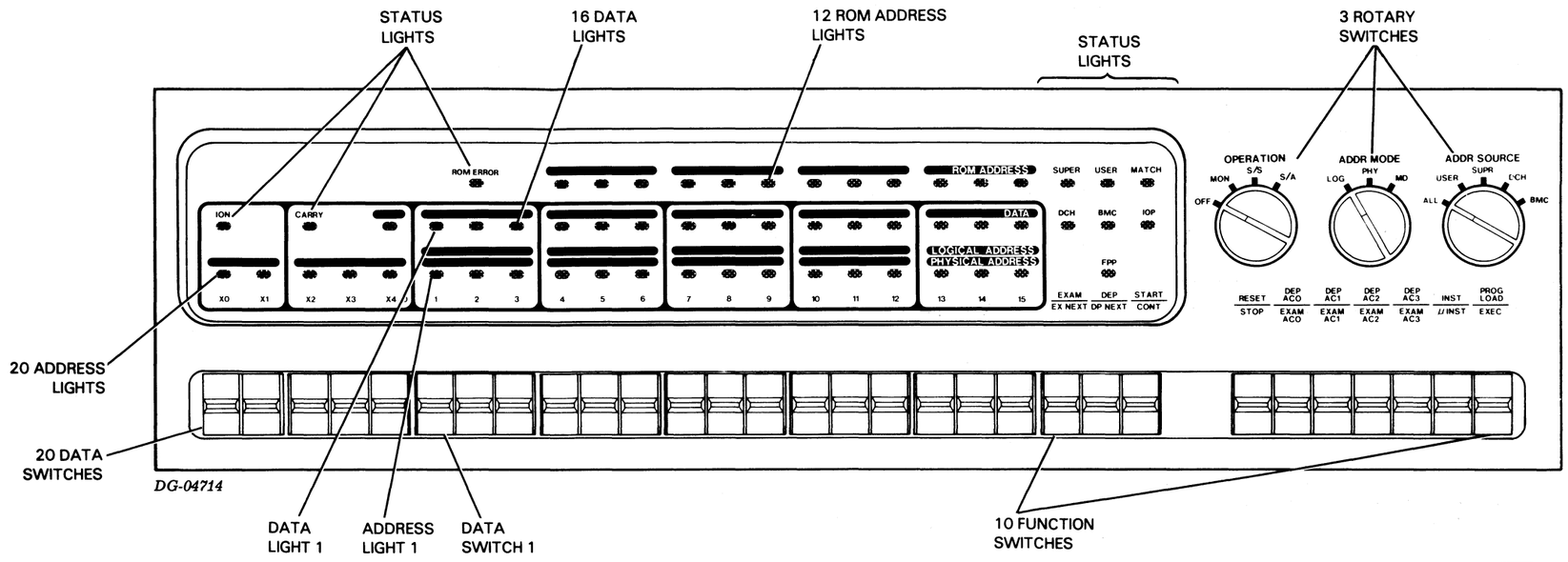
FUNCTION SWITCHES

NAME	POSITION	FUNCTION
Exam/ Exam N	Up	Loads PC with value of data switches, and displays contents of that address. Also fills EA register. To use while processor is running. Operation switch must be set to Mon.
	Down	Increments PC, and displays contents of that address.
Dep/ Dep N	Up	Deposits value in data switches at PC address.
	Down	Increments PC and deposits value in data switches in that address.
St/ Cont	Up	Loads with value in data switches and starts normal execution. Also fills EA register.
	Down	Initiates normal operation from the current state of the machine.
Rest/ Stop	Up	Resets CPU and issues an IORST. ROM address lights display 0002_8.
	Down	Halts the CPU.
AC Dep/ Exam	Up	Loads the associated accumulator with the value in the data switches.
	Down	Displays the contents of the associated accumulator.
Inst/ U Inst	Up	Executes one machine instruction then halts the CPU.
	Down	Freezes the CPU after executing one microinstruction. Address lights show output of ALU.
P Load/ Exec	Up	Loads bootstrap loader program. Data switches 10-15 contain device code. X4/0 is 1 if device is on DCH or BMC, and 4 is 0, if microdiagnostic is to be executed.
	Down	Executes instruction contained in data switches.

STATUS LIGHTS

NAME	MEANING WHEN LIT
ION	I/O Interrupt flag is enabled.
Carry	Carry bit is 1.
ROM Error	Parity error in ROM is detected.*
Super	MAP is in unmapped mode. (MAP B)
User	MAP is in user mode. (MAP A)
Match	The specified address source has access of the address bus.
DCH	Physical address bus is in use by the data channel.
BMC	The BMC is operating.
IOP	Will never light; reserved for future use.
FPP	The floating point processor is performing a floating point operation.

* If a ROM parity occurs, the CPU freezes.



ROTARY SWITCHES

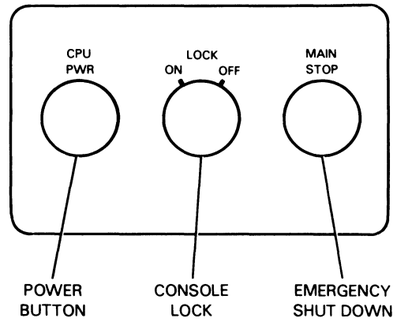
NAME	SETTING	EFFECTS
Operation	Off	Has no effect on processor operation.
	Mon	When address in data switches is accessed, displays contents in data lights.
	S/S	When contents of address in data switches is changed, processor freezes.
	S/A	When address in data switches is addressed, processor freezes.
Addr Mode	Log	Sets logical addressing mode, displays contents of logical address bus in address lights.
	Phy	Sets physical addressing mode; displays contents of physical address bus in address lights.
	MD	Sets memory diagnostics addressing mode; displays contents of physical address bus in address lights.
Addr Source	All	Console monitors all memory addressing sources, except BMC.
	User	Console monitors memory addressing by user MAP (MAP A).
	Super	Console monitors unmapped memory addressing (MAP B).
	DCH	Console monitors memory addressing by the data channel.
	BMC	If the BMC option is included, the console monitors memory addressing by the BMC; otherwise, no monitoring will take place.

ADDRESS AND DATA LIGHTS

NAME	MEANING
ROM Address	Displays the address of the microinstruction last executed.
Data Lights	Displays contents of MEM Bus, except in Monitor mode.
Address	Displays contents of the address bus selected by the Address Mode switch, or the PC when halted.

POWER PANEL

NAME	OPERATION
CPU PWR	Controls DC power to the ECLIPSE S/250 chassis.
LOCK	Enables and disables console switches.
MAIN STOP	Shuts off all power to the cabinet when pushed.



Chapter VIII
CONSOLE FUNCTIONS

The console is a molded plastic panel with lights and switches that display and change the state of the machine. The position on the console and the general function of each of these lights and switches is shown in the removable diagram that precedes this page. There are five types of switches:

- A data switch (also called a toggle switch) -- has two positions. Up corresponds to 1, and down means 0.
- A function switch -- has three positions: up, down and neutral. When pushed up or down, it initiates a function; when released, it returns to the neutral position.
- A rotary switch -- may have any number of positions; once set to a position it remains there until manually altered.
- A button switch -- has one stable position, out. When pushed in, it initiates a function. When released it returns to the stable position.
- A lock -- has two positions and cannot be changed without the key.

Throughout the rest of the section we refer to each of these types of switches by the name given above or by the name of the function that switch performs. However, each *data* switch has its own name (X0-15), which can be seen immediately above it. We use those names to specify some subset of all data switches. The same name also refers to the data light and address light that is immediately above each switch. The console diagram shows the relationship for data light, address light, and data switch 1.

While it is powered up, the CPU is always in one of three states: normal execution, frozen, or halted. When it is in normal execution, the microcode continually executes machine instructions from a program.

When the CPU is frozen, it does not execute microcode and it will not change state without external intervention. While in this state most of the console switches are disabled.

When the CPU is halted, it executes a small microinstruction loop (the ROM address lights display 0002_8, and all of the console switches function normally. The CPU is in the halt state when it is powered up.

MAIN POWER PANEL

NAME	FUNCTION	OPERATION
CPU PWR	DC[1] Power	Controls dc power to the S/250 chassis (does not affect operation of the fans). If the chassis is powered down, pushing this button powers it up; if the chassis is powered up, this button powers it down. When the CPU is first powered up it automatically performs a Reset function.
LOCK	Console Lock	Enables and disables console switches. When in the ON position, auto restart is enabled, only power board switches function; when in the OFF position, auto restart is disabled, and all console switches function.
MAIN STOP	Emergency Shut Down	Shuts off all power to the chassis when pushed. Use it only in the event of an emergency. This button is not disabled by the console lock. Restore AC power by resetting the circuit breakers at the rear of the cabinets.

1 The action of this switch is mechanical; you can turn the switch on and off whether or not ac power is present in the system.

ROTARY SWITCHES

NAME	POSITION	EFFECTS OF SETTING
OPERATION	OFF	Has no effect on processor operation.
	MON	Displays contents of selected location in data lights, if and when that location is accessed. The setting of the data switches specifies the address of the monitored location. Updates the contents of data lights each time that location is accessed. The position of the Address Mode and Address Source switches modify this function. NOTE: Data lights remain unchanged until monitor conditions are met (i.e., the addressing source specified by the Address Source switch reads from or writes to the selected address). If that address is never accessed, the data lights will never display its contents.
	S S	Freezes processor when the contents of the selected location are altered. The setting of the data switches specifies the address of the selected location. Completes the store prior to the freeze. The position of the Address Mode and Address Source switches modify this function.
	S A	Freezes processor when the selected location is accessed. The setting of the data switches specifies the address of the selected location. The location is neither read nor written. The position of the Address Mode and Address Source switches modify this function.
ADDRESS MODE	LOG	Sets console addressing mode to use 15-bit logical addresses. The 15 rightmost address lights (1-15) will display the contents of the logical address bus. When using Monitor, Stop on Store, or Stop on Address to evaluate a program's use of a memory address, use data switches 1-15. NOTE: The Examine and Start functions require a 15-bit logical address in data switches 1-15. The MAP that is active will produce a 20-bit physical address of the location to be examined or executed. However, the 5 leftmost data switches (X0-X4/0) will be used to fill the EA register. (For more detail see Console Section, Chapter II.)
	PHY	Sets console addressing mode to use 20-bit physical addresses. All 20 address lights will display the contents of the physical address bus. When using Monitor, Stop on Store, or Stop on Address to evaluate a program's use of a memory address, use all 20 data switches. NOTE: The Examine and Start functions require a 20-bit physical address in data switches X0-15. The 5 leftmost Data switches (X0-X4/0) will be used to fill the EA register. (For more detail see Console Section, Chapter II.)
	MD	Sets console addressing mode to memory diagnostics. All 20 address lights will display the contents of the physical address bus. When using Monitor, Stop on Store, or Stop on Address to evaluate a program's use of a memory address, use all 20 data switches. NOTE: The Examine and Start functions require a 20-bit physical address in data switches X0-15. The 5 leftmost data switches will fill the EA register. The MAP is inactive, and neither the console nor executing programs use it to generate physical addresses. (For more detail see the Console Section, Chapter II.)

ROTARY SWITCHES

NAME	POSITION	EFFECTS OF SETTING
ADDRESS SOURCE	ALL	Specifies that the console will monitor all memory addressing sources except BMC.
	USER*	Specifies that the console will monitor memory addressing by the user (Map B).
	SUPER*	Specifies that the console will monitor memory addressing by the supervisor (Map A).
	DCH	Specifies that the console will monitor memory addressing by the data channel.
	BMC	If the BMC option is included, specifies that the console will monitor addressing by the burst multiplexor channel. NOTE: Since the Burst Multiplexor does not use the logical address bus, when the Address Source is set to BMC the Address Mode switch must be set to PHY or MD for any monitoring to occur. The BMC cannot be monitored for a Stop on Store or a Stop on Address. When the processor freezes due to a Stop on Store or a Stop on Address, the BMC will not necessarily stop reading or writing memory.

* In memory diagnostic mode the MAP must be inactive so the User and Supervisor distinctions will not exist. If the Address Source switch is set to User or Supervisor and the Address Mode switch is set to MD, no monitoring should occur.

FUNCTION SWITCHES

NAME	POSITION	FUNCTION	MACHINE STATE*	MEANING
EXAM/ EX NEXT	UP	EXAMINE	HALTED	Loads PC with the logical address contained in data switches 1-15. Displays contents of that location in data lights, and displays address of that location in address lights.
			RUNNING	Displays contents of memory at location addressed by data switches. The Operation switch must be set to Monitor or Stop on Store for the display to remain long enough to be read. A running examine will not change the PC. NOTE: The Examine function also fills the EA register with the value contained in the 5 leftmost data switches (X0-X4/5). This register is used when the Address Mode switch is set to MD or PHY. (For more detail see Console Section, Chapter II.)
	DOWN	EXAMINE NEXT	HALTED	Increments PC, and uses that number as an address. Displays the contents of that address in data lights. Displays address of that location in address lights.
DEP/ DP NEXT	UP	DEPOSIT	HALTED	Stores the value contained in the 16 rightmost data switches (X4/0-15) into the location addressed by PC. Displays new value of that location in data lights, and displays address of that location in the address lights.
	DOWN	DEPOSIT NEXT	HALTED	Increments PC and uses that number as an address to store value contained in the 16 rightmost data switches (X4/0-15). Displays new value of that location in data lights, and displays address of that location in address lights.
START/ CONT	UP	START	HALTED	Loads the contents of the 15 rightmost data switches into PC, and executes the instruction at that address. Normal execution continues from there. Displays the last contents of the memory bus in data lights, and displays the contents of the selected address bus in address lights. NOTE: The Start function also fills the EA register with the value contained in the 5 leftmost data Switches (X0-X4/0). This register is used when the Address Mode switch is set to MD or PHY. (For more detail see Console Section, Chapter II.)
	DOWN	CONTINUE	HALTED, FROZEN	Initiates normal operation of the CPU from the current state of the machine.
RESET/ STOP	UP	RESET	RUNNING, FROZEN, HALTED	Stops the CPU immediately, initiates the equivalent of an I/O Reset instruction, setting the Busy and Done flags of all peripherals to 0. Sets all status lights on the console, except Carry, to 0. The ROM address lights will display 0002_8 (the halt location). The contents of the data and address lights are undefined. NOTE: The PC is unchanged; however, the instruction addressed by the current PC value may not have completed execution. This is the only function switch that will halt the CPU in the middle of an instruction.
	DOWN	STOP	RUNNING	Halts the CPU after the current instruction has been executed. Displays the address of the next instruction to be executed in address lights. Displays the last contents of the memory bus in data lights. The ROM address lights will show 0002_8 (the halt location). NOTE: Data channel requests will be honored after the halt, and the BMC will continue to access memory. But interrupt requests will not be honored after the Stop function has been initiated.

FUNCTION SWITCHES

NAME	POSITION	FUNCTION	MACHINE STATE*	MEANING
DEP AC/ EXAM AC**	UP	DEPOSIT	HALTED	Loads the associated accumulator with the value contained in the 16 rightmost data switches (X4/0-15). Displays the new contents of the AC in data lights.
	DOWN	EXAMINE	HALTED	Displays the contents of the associated accumulator in data lights.
INST/ uINST	UP	STEP INSTRUCTION	HALTED, FROZEN, RUNNING	Executes one machine instruction; then halts the processor. Displays the contents of the memory bus in data lights, and displays the address of the next instruction to be executed in address lights. (See the section on debugging through the console, Chapter II.)
	DOWN	STEP MICRO-INSTRUCTION	HALTED, RUNNING	Executes one microinstruction; then freezes the CPU. Displays the contents of the MEM bus in the data lights; displays the output of the ALU bus in the address lights. Displays the address of the last microinstruction executed in the ROM address lights. (See the section on debugging through the console, Chapter II.)
PROG LOAD/ EXEC	UP	BOOTSTRAP LOAD	HALTED	Executes a microdiagnostic program; then loads bootstrap loader program into memory locations $0\text{-}37_8$, and executes it. If data switch 4 is 1 microdiagnostic will not be executed. Data switches 10-15 must contain the device code of the I/O device that contains the program to be loaded. If that device is on the data channel or the burst multiplexor channel, data switch (X4/0) must be set to 1. (See the discussion of bootstrap loading in Chapter II.)
	DOWN	EXECUTE	HALTED	Executes instruction contained in 16 rightmost data switches (X4/0-15), and halts the CPU. (Execute may be used with step microinstruction -- see discussion of debugging through the console in Chapter II.) NOTE: PC will be updated but the instruction at the old PC address will not be executed.

* If a function definition has no entry for a particular machine state, that function has no effect when in that state.

** There are 4 AC Dep/Exam switches on the S/250 console. Each performs the same functions on a different accumulator.

OPERATION MONITORING CONDITIONS

OPERATION: OFF
ADDR MODE: LOG, PHY, MD
ADDR SOURCE: ALL, USER, SUPER, DCH, BMC

The data lights display the contents of the memory bus. Depending on the addressing mode, the address lights will show the contents of either the physical or logical address bus. The Address Source switch has no effect for this operation.

OPERATION: MON
ADDR MODE: LOG*
ADDR SOURCE: ALL, USER, SUPER, DCH

The data lights display the contents of the memory bus when the contents of the logical address bus match the address contained in the data switches. The 15 rightmost data switches (1-15) specify the logical address of the monitored locations. The data lights are updated each time an address match is detected, and they remain unchanged between matches. The address lights display the last contents of the logical address bus. The Address Source switch specifies the originator of the access.

OPERATION: MON
ADDR MODE: LOG
ADDR SOURCE: BMC

The burst multiplexor channel never accesses the logical address bus. The data lights will never change while the rotary switches remain in these positions.

OPERATION: MON
ADDR MODE: PHY
ADDR SOURCE: ALL, USER, SUPER, DCH, BMC ***

The data lights display the contents of the memory bus each time the contents of the physical address bus match the address contained in the data switches. All 20 data switches specify the physical address of the single monitored location. The data lights are updated each time an address match is detected, and they remain unchanged between matches. The address lights display the last contents of the physical address bus. The Address Source switch specifies the originator of the access.

OPERATION: MON
ADDR MODE: MD**
ADDR SOURCE: ALL, DCH, BMC ***

The data lights display the contents of the memory bus each time the contents of the physical address bus match the address contained in the data switches. All 20 data switches specify the physical address of the single monitored location. The data lights are updated each time an address match is detected, and they remain unchanged between matches. The address lights display the last contents of the physical address bus. The Address Source switch specifies the originator of the access.

OPERATION: MON
ADDR MODE: MD
ADDR SOURCE: USER, SUPER

The MAP should be disabled in MD mode. While the rotary switches remain in this setting, no monitoring should take place.

OPERATION: S/S
ADDR MODE: LOG*
ADDR SOURCE: ALL, USER, SUPER, DCH

The processor freezes when the contents of the logical address bus match the address contained in the data switches during a write operation. The 15 rightmost data switches (1-15) specify the logical address that will cause the freeze. The location is written prior to the freeze, and the Match lamp lights to indicate the cause of the freeze. The address lights display the last contents of the logical address bus. The Address Source specifies the originator of the store.

OPERATION: S/S
ADDR MODE: LOG, PHY, MD
ADDR SOURCE: BMC

The burst multiplexor channel may not be used as the address source for a Stop on Store. While the switches remain in this setting, the processor will never freeze due to a Store by the BMC.

OPERATION: S/S
ADDR MODE: PHY
ADDR SOURCE: ALL, USER, SUPER, DCH

The processor freezes when the contents of the physical address bus match the address contained in the data switches during a write operation. All 20 data switches specify the physical address that will cause the freeze. The location is written prior to the freeze, and the Match lamp lights to indicate the cause of the freeze. The address lights display the last contents of the physical address bus. The Address Source specifies the originator of the write.

OPERATION: S/S
ADDR MODE: MD**
ADDR SOURCE: ALL, DCH

The processor freezes when the contents of the physical address bus match the address contained in the data switches during a write operation. All 20 data switches specify the single physical address that will cause the freeze. The location is written prior to the freeze, and the Match lamp lights to indicate the cause of the freeze. The address lights display the last contents of the physical address bus. The Address Source switch specifies the originator of the write.

OPERATION: S/S
ADDR MODE: MD
ADDR SOURCE: USER, SUPER

The MAP must be disabled in MD mode. While the the rotary switches are in this setting the processor should never freeze due to a Stop on Store.

OPERATION: S/A
ADDR MODE: LOG*
ADDR SOURCE: ALL, USER, SUPER, DCH

The processor freezes when the contents of the logical address bus match the address contained in the data switches. The 15 rightmost data switches (1-15) specify the logical address that will cause the freeze. The location is not read or written, and the Match lamp lights to indicate the cause of the freeze. The address lights display the last contents of the logical address bus. The setting of the Address Source switch specifies the originator of the access.

OPERATION: S/A
ADDR MODE: LOG, PHY, MD
ADDR SOURCE: BMC

The burst multiplexor channel is not monitored for a Stop on Address operation. While the rotary switches are in this setting, the processor will never freeze due to an access by the BMC.

OPERATION: S/A
ADDR MODE: PHY
ADDR SOURCE: ALL, USER, SUPER, DCH

The processor freezes when the contents of the physical address bus match the address contained in the data switches. All 20 data switches specify the physical address that will cause the freeze. The location is not read or written, and the Match lamp lights to indicate the cause of the freeze. The address lights display the contents of the physical address bus. The Address Source switch specifies the originator of the access.

OPERATION: S/A
ADDR MODE: MD**
ADDR SOURCE: ALL, DCH

The processor freezes when the contents of the physical address bus match the address contained in the data switches. All 20 of the data switches specify the single physical address that will cause the freeze. The location is not read or written, and the Match lamp lights to indicate the cause of the freeze. The address lights display the last contents of the physical address bus. The Address Source switch specifies the originator of the access.

OPERATION: S/A
ADDR MODE: MD
ADDR SOURCE: USER, SUPER

The MAP should be disabled in MD mode. While the rotary switches are in this setting, the processor should never freeze due to a Stop on Address.

* A single logical address may specify several physical locations in memory. When in logical addressing mode, several mapped locations may be monitored simultaneously. (See section on the MAP in Chapter II for details.)

** In memory diagnostic mode only 32K of memory is addressable because the MAP is turned off. The burst multiplexor channel, since it uses its own MAP can address the entire memory even in MD mode. (See section on the console in Chapter II for more details.)

*** If the BMC option is not included in your system, then no monitoring will take place.

APPENDIX A
HOST/SP COMMUNICATION

This appendix discusses the ECLIPSE S/250 facilities that support communications between the SP and the host. In the following discussions, we will use the term SP when describing a feature common to both the 8660 SP and the 8661 SP. We will use the specific terms *8660 SP* and *8661 SP* when describing features whose specifications differ between the processors.

Overview

The SP is a complete ECLIPSE processor that resides within the chassis of the host ECLIPSE S/250. It features standard ECLIPSE facilities such as stack, standard I/O bus, ECLIPSE instruction set (the 8660 SP includes the character instruction set as well), priority interrupt system, etc. The SP is user programmed.

The 8660 SP

The 8660 SP can be programmed to handle much of the text editing chores of a system, thereby leaving the host free for efficient task management and other administrative jobs. A multiple 8660 SP system can be programmed to use the host as an intercommunications network between the 8660 SPs, with each 8660 SP handling its own group of I/O devices. 8660 SPs can perform any other communications function normally carried out by an ECLIPSE processor, including message concentration, network control, and real-time processing and control.

The 8661 SP

Since the 8661 SP can manipulate complex numbers and arrays, it can be programmed to handle situations where a large amount of data has to be processed in a short amount of time. Many scientific applications, such as signal processing, fall into this area. The 8661 SP's ability to handle such jobs easily means that time will be saved, and the host can devote time to other tasks.

Forms of Host/SP Communication

Divisions of labor such as those described above require some form of communication between the host processor and the SP. Communication is necessary to coordinate the operation of the two processors; for example, the SP must be able to signal the host when it has completed a task, or needs more information. The SP MAP and two groups of special instructions provide the host and the SP with the necessary ability to communicate.

The MAP

The SP hardware MAP allows the SP to communicate directly with host memory. The MAP intercepts memory references made by the SP and determines whether the reference is to a local SP memory location or a host memory location. When the reference is to host memory, the MAP sends the reference along the host data channel and through the host map to the correct memory location. This means that the SP has direct access to information stored in the host.

Communication Instructions

The host uses more indirect means to manipulate the SP. A group of special instructions allow the host to access SP memory in much the same way as you would use control panel switches. With these instructions the host can load the SP map, check the map status, and manipulate the contents of various SP registers. This allows the host to oversee the general operation of the SP.

The SP also has a special group of instructions which allow it to modify some aspects of the host's operation. While they do not give the SP supervisory control over the host, they do allow the SP to change the host data channel map, and determine where information will be loaded into host memory.

Each of the host's and the SP's special instructions contains an optional group of mnemonics that you can use to manipulate host and SP flags. These flags are the Busy, Done, and interrupt request flags and they are used to indicate the state of the processors. By

setting some of its flags the host or the SP can signal the continuation or completion of a task or request an interrupt.

Elements of the SP

The SP is a self-contained ECLIPSE CPU. This means that the SP includes the following:

- the ECLIPSE stack
- standard ECLIPSE instruction set
- standard NOVA/ECLIPSE data channel for medium- to high-speed devices
- programmed I/O with priority interrupt handling and vectoring capability
- extended operation feature
- arithmetic logic unit

SP Memories

The 8660 SP contains 64 Kbytes of local semiconductor memory. It is not expandable. Word length is 16 bits. The address range is from 0 to 77777_8.

The 8661 SP contains 56 Kbytes of local semiconductor memory and 8 Kbytes of AP memory. Expansion capability, word length, and address range are identical to those of the 8660 SP.

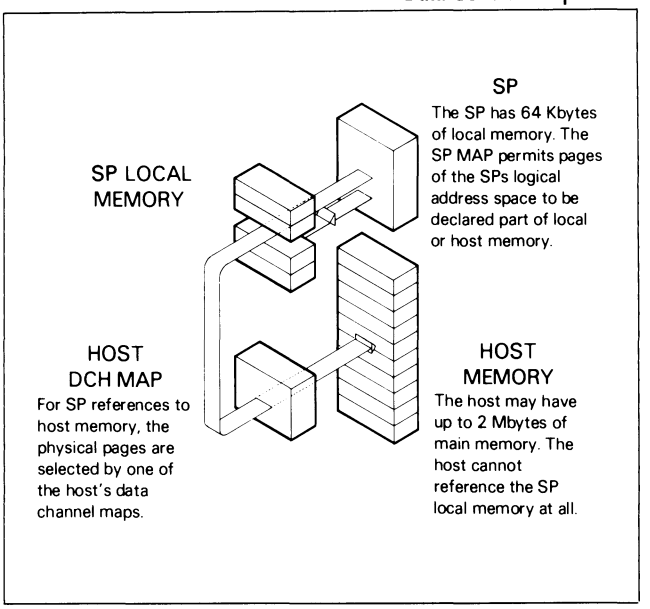
MAP

The SP MAP can map 2 Kbyte pages of SP address space into either SP local memory or host memory. To do this, the MAP contains two sets of pointers for SP data channel and program memory references. Each set contains one pointer for each page. When either type of memory reference occurs, the five most significant bits of the logical address select a 1-bit pointer from the MAP. This pointer determines if the address is to local memory or host memory. In references to local memory, the unaltered logical address references some local memory location. In references to host memory, the address is sent across the host data channel to the host MAP. The host MAP interprets the physical location of the address sent by the MAP.

Note that for the 8661 SP, hardware considerations allow only the 56 Kbytes of SP local memory to be mapped to the host. Host I/O, however, can address the combined 64 Kbytes of SP local memory and AP memory as one contiguous address space.

User and Data Channel Maps

The SP has both a user map and data channel map supported on its own I/O bus. The SP uses the *SP* data channel map when transferring data to SP I/O devices. The SP uses the *host* data channel and *host* data channel maps when referencing host memory.



Parity Generator

The SP parity generator provides a parity bit for every word written into SP local memory. To enable parity checking for SP local memory, use the *Read Map Status and Parity Control* instruction. This instruction will set the Parity Enable bit (bit 4) in the map status and parity control register. Detection of an error will reset the SP and its devices, and will set the host interrupt request flag.

Host-SP Interface

The host-SP Interface is a group of registers in the SP that functions as a communications link between the SP and the host. These registers serve two purposes:

- They act as the console of the SP. Any function normally performed on ECLIPSE front panel console switches can be performed using these registers (except micro-instruction step).
- They can be used by the host to control and monitor the SP. The host does this by reading information from or writing information to these registers.

Interface Elements

A description of the interface's main elements and the host instructions which access them is given below. Unless otherwise specified, the registers are host-accessible via programmed I/O only.

Console Switch Register - This register takes the place of the data switches on a standard console. Load this register from the host using a *Load Console Switch Register* instruction.

Console Function Register - This register takes the place of the function switches on a standard console. Load this register using a *Load Console Function Register* instruction.

Address Register - This register holds the address of the location last referenced by the SP. Use the *Read Address Register* to examine the contents of this register.

PC Save Register - Each time the SP halts, the value of the PC is stored in this register. Use the *Read PC Save Register* instruction to examine the contents of this register.

Console Register - This register serves the same purpose as the data lights on a standard console. Use the *Read Console Buffer* instruction to examine the contents of this register.

CROSS INTERRUPTS

When the SP needs information from the host, it signals the host for an interrupt by setting the host's *interrupt request flag.* The host checks this flag for interrupts at the end of each memory cycle. If an interrupt is pending, then the host halts its program execution long enough to service the interrupt and set its interrupt request flag to 0.

Similarly, when the host needs information from the SP, it signals the SP for an interrupt by setting the SP's interrupt request flag.

Setting the Interrupt Request Flags

To set the interrupt request flags, use the appropriate flag control command appended to the mnemonic of one of the host I/O instructions (see Chapter VI) or one of the SP I/O instructions (see Chapter VII). The specific ways to set the interrupt request flags are described below.

The host can set the SP's interrupt request flag by issuing an **S** flag control command to the SP device code. This can be cleared by issuing a **C** flag control command from the SP to device code 4.

The SP can set the host's interrupt request flag by issuing an **S** flag control command to device code 4. This can be cleared by issuing a **C** flag control command from the host to the SP device code. Note that a parity error in SP local memory will also cause a host interrupt and SP system reset. This interrupt can be cleared by issuing a **P** flag control command from the host to the SP.

The methods described above will set other flags at the same time they set the interrupt request flags. These other flags are listed below, with tables of the flag control commands and their actions.

Busy and Done Flags - Both the host and the SP have a Busy flag and a Done flag which indicate the state of the processor. The SP's Busy and Done flags are located in the host; the host's Busy and Done flags are located in the SP. Setting the SP's Done flag to 1 causes an interrupt in the host. Setting the host's Busy flag to 1 will cause an interrupt in the SP. The SP's Busy flag is set to 1 whenever the SP is running.

Parity Error Flag - The SP parity generator sets this flag to 1 when it finds a parity error. If the SP does not mask out parity errors, this flag generates a host interrupt when set to 1. See the *Control Map and Page/Parity* instruction in Chapter VI for more information about this process.

The table below lists the instructions used by the host to communicate with the SP.

Host Communications Instructions

Mnem	Name	Function
DIA SP0	Read PC Save Register	Loads the contents of the SP PC save register into an accumulator.
DIB SP0	Read Console Buffer	Loads the contents of the SP console buffer into an accumulator.
DIC SP0	Read Address Buffer	Loads the contents of the SP address buffer into an accumulator.
DOA SP0	Control Console Function Register	Stores the contents of an accumulator into the SP console function register.
DOB SP0	Control Switch Register	Stores the contents of an accumulator into the SP switch register.

The following lists the flag control commands used with the host communications instructions.

f=**S** Sets the host Busy and the SP interrupt request flags to 1

f=**C** Sets the SP Done and host interrupt request flags to 0

f=**P** Sets the SP parity error and host interrupt request flags to 0

IORST Sets the SP Done flag, host interrupt request flag and host interrupt mask bit to 0; also resets the SP processor and its I/O devices

NOTE: *The* **C** *flag control command does not clear a host interrupt caused by an SP parity error. The* **P** *flag control command does not clear a host interrupt caused by the SP Done flag.*

The next table lists the instructions used by the SP to communicate with the host.

SP Communications Instructions

Mnem	Name	Function
DIA ISP0	Read Map Status	Loads the map status and parity control bits and MAP host/local flag into an accumulator.
DOA ISP0	Control Map and Parity	Stores the contents of an accumulator into the map status and parity control register.
LMP	Load Map	Loads a number of map entries from a table in memory to the SP MAP.

The last lists the flag control commands used with the SP instructions.

f=S Sets the SP Done and host interrupt request flags to 1

f=C Sets the host Busy and SP interrupt request flags to 0

f=P Sets the host Done flag to 0

IORST Sets bits 2-4, 14 and 15 of the map status and parity control register, host Done flag, SP interrupt request flag, host Busy flag, and SP interrupt mask bits to 0

PROGRAMMING EXAMPLES

We include two programming examples to illustrate host/SP communications. The first example shows you how to load data into the SP. The second example is more complex; it shows you how to start and load the SP.

Example 1

Since there are no console switches on the SP, you must issue special instructions from the host in order to place information in the SP. These special instructions use the SP *console registers* to move data from the host to the SP.

To place a piece of data into an SP local memory location, first load the SP address receiving the data into the switch register. Then specify the Examine function by loading the function register with the numerical representation of Examine. Next, load the data value into the switch register, and load the numerical representation of the Deposit function into the function register.

This example shows this method of loading data into the SP. See Ch. 5 for a discussion of the host instructions used below, plus a table of numerical representations of the functions.

```
                .
                .       ; previous part of program
                .

                        ; this part of the program will load two
                        ; words of data from the host into SP
                        ; locations 100 and 101

C0:     0               ; constant 0
C100:   100             ; this is the address of the SP
                        ; location
                        ; that will receive the first data word

EX:  050000             ; code for the Examine function
DP:     040000          ; code for the Deposit function
DN:     044000          ; code for the Deposit Next function
DATA:   LDA     1,C0    ; these are the two words that will be
        STA     1,STR   ; loaded into the SP
        LDA     0,C100
        DOB     0,SP0   ; put addr of SP location in data
                        ; switches
        LDA     1,EX
        DOA     1,SP0   ; load console function register with code
                        ; of Examine function
        LDA     0,DATA
        DOB     0,SP0   ; put first data word in data switches
        LDA     1,DP
        DOA     1,SP0   ; load console function register with
                        ; code of Deposit function
        LDA     0,DATA+1
        DOB     0,SP0   ; put second data word into data switches
        LDA     1,DP
        DOA     1,SP0   ; load console function register with code
                        ; of deposit next function and deposit
                        ; next word into location 101


                .       ; rest of program
                .
```

EXAMPLE 2

This example shows how to start the SP. The program below is put into host memory, starting at location 30. It loads a small bootstrap program and map data into SP locations 0-53 via the SP console registers. Once it loads the bootstrap, it transfers control to the bootstrap. The bootstrap then remaps the SP and, using the **BLM** instruction, transfers the contents of 8K of host memory locations starting at location 40000 to the first 8K of SP memory.

```
; This program demonstrates the use of the host/SP communications
; facilities. The host program places a 43 word bootstrap program
; in the SP via the console functions and starts the SP.The
; SP then moves 8K words into its local memory from host memory
; using its mapping capability.
        .TITL   START
        .TXTM   1
                                ; AC0 holds switch data for SP
                                ; AC1 holds function data for SP
                                ; AC2 holds array address for SP
boot
                                ; AC3 unused
START:  INTDS
        SUB     0,0
        DOB     0,SP0           ; deposit zero in data switches (this
                                ; will be the starting location of the
                                ; boot program in the SP)
        LDA     1,EX
        DOA     1,SP0           ; specify the examine function
        LDA     2,.A
        LDA     0,0,2           ; addr of SP bootstrap in host
        DOB     0,SP0           ; deposit word 1 of boot in switches
        LDA     1,DP
        DOA     1,SP0           ; specify deposit function
        INC     2,2
        LDA     1,DN            ; load deposit next fn in AC1 for loop
LOOP:   LDA     0,0,2
        DOB     0,SP0           ; deposit next word of boot in switch
        DOA     1,SP0           ; specify deposit next function
        INC     2,2
        DSZ     CNT             ; test for last word of SP boot
        JMP     LOOP
        LDA     0,C40           ; load # of words loaded in SP LMP
        DOB     0,SP0           ; deposit # of words in switches
        LDA     1,DP1
        DOA     1,SP0           ; specify deposit into AC1 function
        LDA     0,IMAP
        DOB     0,SP0           ; load SP map data adr into switches
        LDA     1,DP2
        DOA     1,SP0           ; specify deposit into AC2 function
DONE:   SUB     0,0
        DOB     0,SP0           ; deposit zero into switches (where to
                                ; start execution in SP)
        LDA     1,STR
        DOA     1,SP0           ; specify start function
.A:     A+1
CNT:    54                      ; # words in SP boot
EX:     050000                  ; examine function
DP:     040000                  ; deposit function
DN:     044000                  ; deposit next function
STR:    060000                  ; start function
C40:    40                      ; # words in SP LMP
DP1:    024000                  ; deposit in AC1 function
DP2:    030000                  ; deposit in AC2 function
IMAP:   10                      ; address of SP MAP data
                                ; This bootstrap is loaded into the
                                ; SP by the preceding host program
                                ; via the SP console functions. The
                                ; SP then remaps itself so that its
                                ; upper 16K address space is mapped into
                                ; the host. The SP then moves 8K of
                                ; data from the host to its lower 8K,
                                ; overwriting the bootstrap.
A:      (A+1)
        DICC    0,CPU           ; reset SP
        LMP                     ; load the map
        LDA     0,MAPE
        DOA     0,ISP0          ; load map status
        LDA     1,K8            ; # words to move in BLM
        LDA     2,ACS           ; location in Host of data to be moved
        LDA     3,ACD           ; location in SP to put data
        BLM                     ; move the data
MDATA:  000000                  ; The SP map data
        002000
        004000
        006000
        010000
        012000
        014000
        016000
        020000
        022000
        024000
        026000
        030000
        032000
        034000
        036000
        140000
        142000
        144000
        146000
        150000
        152000
        154000
        156000
        160000
        162000
        164000
        166000
        170000
        172000
        174000
        176000
ACD:    0                       ; Destination of SP BLM.
ACS:    40000                   ; Source of SP BLM.
K8:     17777                   ; # words moved by BLM.
MAPE:   100001
        .LOC    040000
                                ; This area of host memory contains the
                                ; 8K of data to be moved into the SP.
                                ; After control is transferred to the
                                ; SP the programs contained in this
                                ; data will begin execution.
        .END    START
```

APPENDIX B
INTEGRAL ARRAY PROCESSOR

ARRAY PROCESSING

Integral Array Processor

Some computing applications, notably scientific programming, are characterized by operations that must be performed on each element of a large data array. General purpose computers solve this problem with program loops, which have the disadvantage of requiring that the instructions in the loop body be repeatedly fetched and decoded. The Integral Array Processor (IAP) enables a single instruction to perform a series of operations, e.g. adding two arrays of numbers and storing the results in a third. The elimination of all the additional instructions results in a substantial improvement in execution speed.

In general, programming the IAP consists of setting up a *parameter block* and executing an IAP instruction. All memory used by the IAP (except the scratchpad) is in the address space of the user's program. Furthermore, the IAP instructions are executed by the host CPU; the IAP is an extension of the host's instruction set, rather than a separate processor. These two facts make IAP programming a relatively simple matter.

The parameter block is a group of 16 consecutive words in the user's address space which control the operation of the IAP. The IAP starts executing each instruction by loading various words from this block. These words specify the starting addresses and sizes of the arrays to be operated on. Additional words of the block control the IAP's response to error conditions such as overflow. The format of the parameter block is described in detail later in this chapter.

The IAP performs all calculations on an element by element basis. For instance, when adding two arrays, the IAP takes one element from each array, adds them, and stores the result in memory before fetching the next pair of elements. This means that data and result arrays may *overlap*; that is, the result array may overwrite part or all of the data arrays. Of course, you must be sure that no data is overwritten before it is used; for this reason overlapping is not recommended except if you need to optimize memory space.

Another advantage of the IAP's element by element operation is that interrupts may occur during the execution of an IAP instruction. This ensures fast response to interrupts, which keeps the IAP from placing a heavy load on the system in which it is installed.

IAP Data Organization

Scalars

The IAP operates on three different types of numbers. The term *scalar* is used throughout this manual to mean single numbers (as opposed to arrays) which are stored in memory or other registers for use by the IAP. Instructions are available to convert data from one type to another. The various types are described below.

Integers

Integers are 16-bit binary numbers in two's-complement format (the IAP does not use double-precision integers). See the Number Convention section in this chapter for more information about integers.

Real scalars

Real scalars are 32-bit single precision normalized floating point numbers in standard Eclipse format. A real scalar occupies two consecutive words of memory. See the Number Convention section in this chapter for more information about floating point numbers.

Complex scalars

Complex scalars are ordered pairs of real scalars. The first number is the *real* part, and the second is the *imaginary* part. A complex scalar is 64 bits long, and is stored in four consecutive words of memory.

Note that there is no explicit difference between a complex scalar and two consecutive real scalars. The only difference is in how the IAP handles the data.

Structure of arrays

An IAP array is an ordered sequence of scalars. The term *elements* is used throughout this manual to refer to the individual scalars -- integer, real, or complex -- which make up an array. All elements of an IAP array must be of the same type; e.g., an array cannot contain a mixture of real and integer elements.

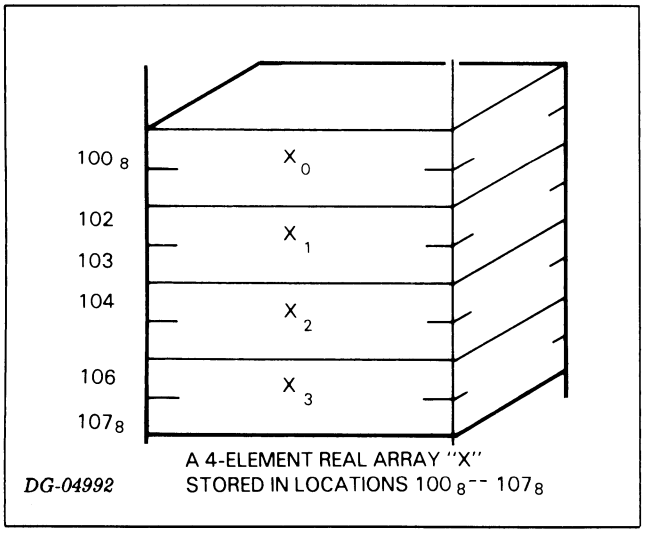
Arrays are usually stored in a block of consecutive memory locations. The elements of a one-dimensional array may simply be ordered sequentially. Multi-dimensional arrays may be represented by splitting them into rows and ordering the rows in memory.

Indices

The IAP interprets some addresses as pointers to *elements,* not memory words. For example, when processing real numbers, the IAP interprets an address of 4 as a pointer to the 4th real scalar of the array, which is contained in the 8th and 9th words. When processing complex numbers, the IAP interprets an address of 4 as a pointer to the complex scalar in words 16-19 of the array.

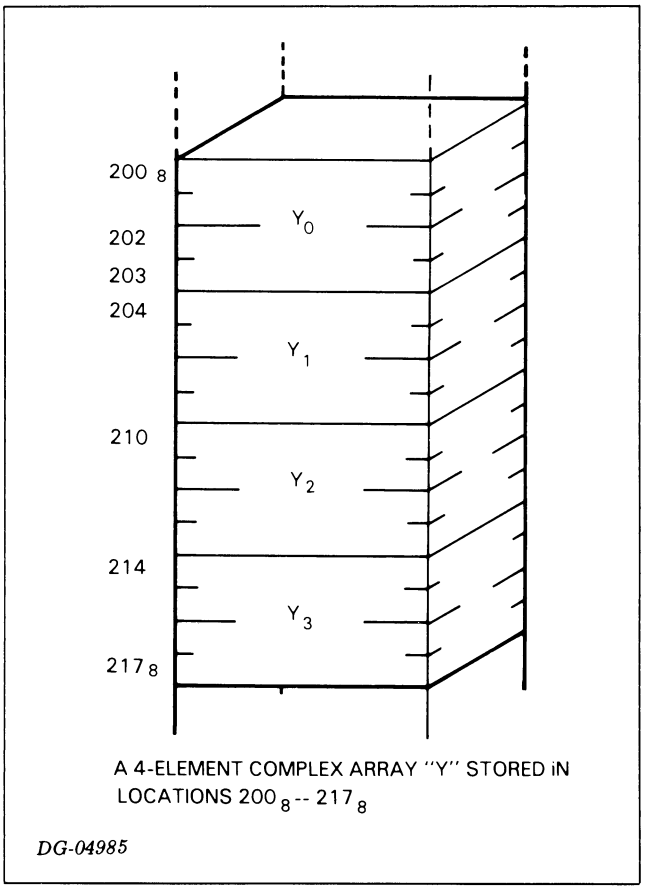
> **NOTE:** *The term* index *is used throughout this manual to indicate addresses whose exact meaning is dependent on the data type as described in the preceding paragraph.*

The accompanying diagrams show the format of several typical arrays in memory.



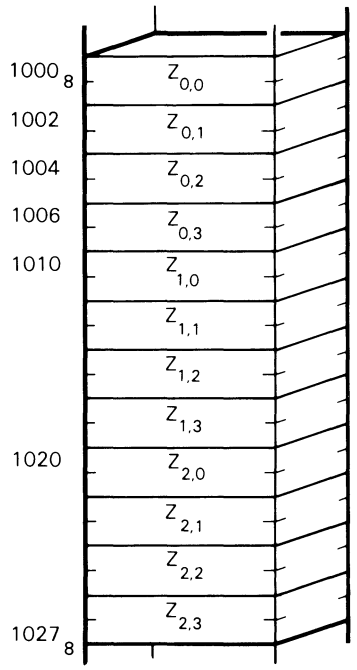
A 4-ELEMENT REAL ARRAY "X"
STORED IN LOCATIONS 100_8-- 107_8

DG-04992



A 4-ELEMENT COMPLEX ARRAY "Y" STORED IN
LOCATIONS 200_8-- 217_8

DG-04985



A 3 * 4 ELEMENT REAL ARRAY
"Z" STORED IN LOCATIONS
1000_8--1027_8

DG-04988

Initial conditions

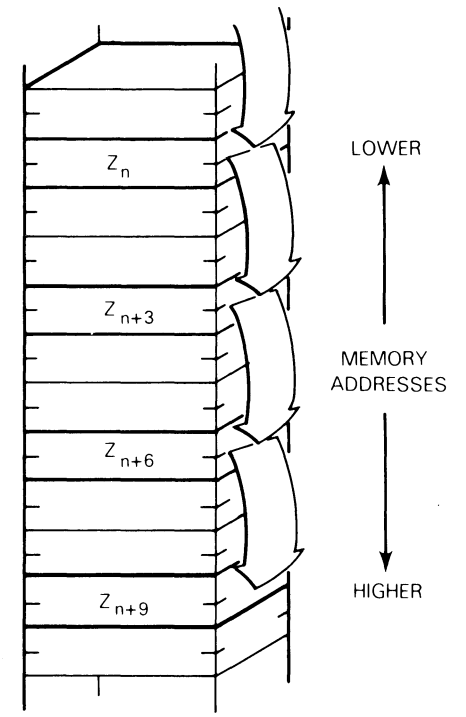
Generally all arrays start with the 0th element. Certain instructions, however, require *initial conditions* which are represented by elements with negative subscripts. When using an array with initial conditions, you should still place the index of the 0th element in the parameter block.

Address step registers

Three registers in the IAP store the *step values* used to reference successive elements of an array. The contents of one of these registers is added to the current index before storing or loading each element of an array. Thus the IAP can reference every *n*th element in memory, where *n* is the value of the register. Note that *n* may be negative to specify that successive elements are stored in descending memory addresses. *n* may also be 0, in which case a single element of the array is used repeatedly.

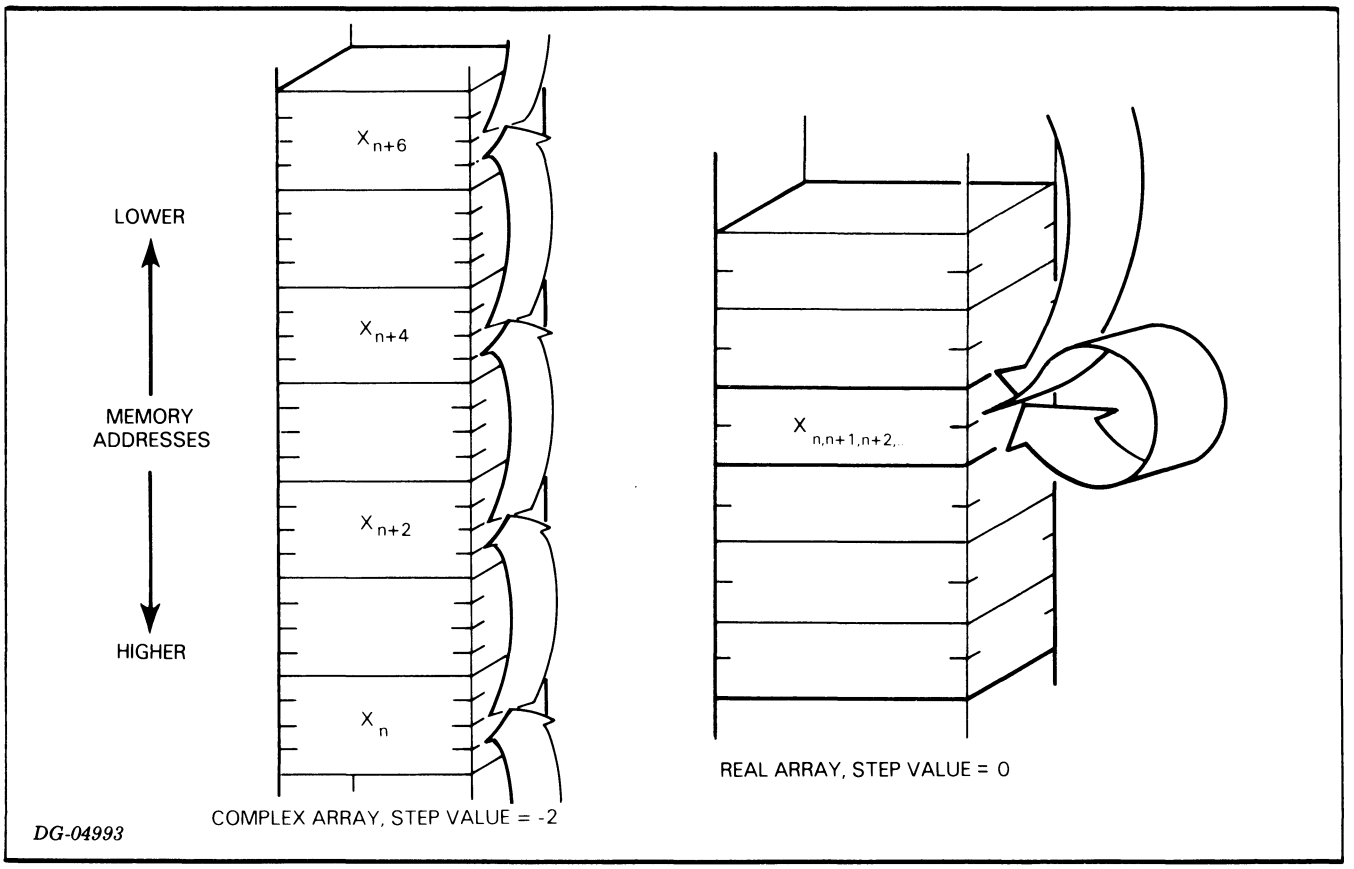
The address step registers are 12 bits long. Instructions that operate on more than one array usually have different step registers for each array. The accompanying diagrams illustrate the use of address stepping.

The address step registers greatly increase the power of the IAP for many types of calculations. Typical uses of address stepping include column-wise operations on multi-dimensional arrays, and extraction of real or imaginary parts from a complex array.



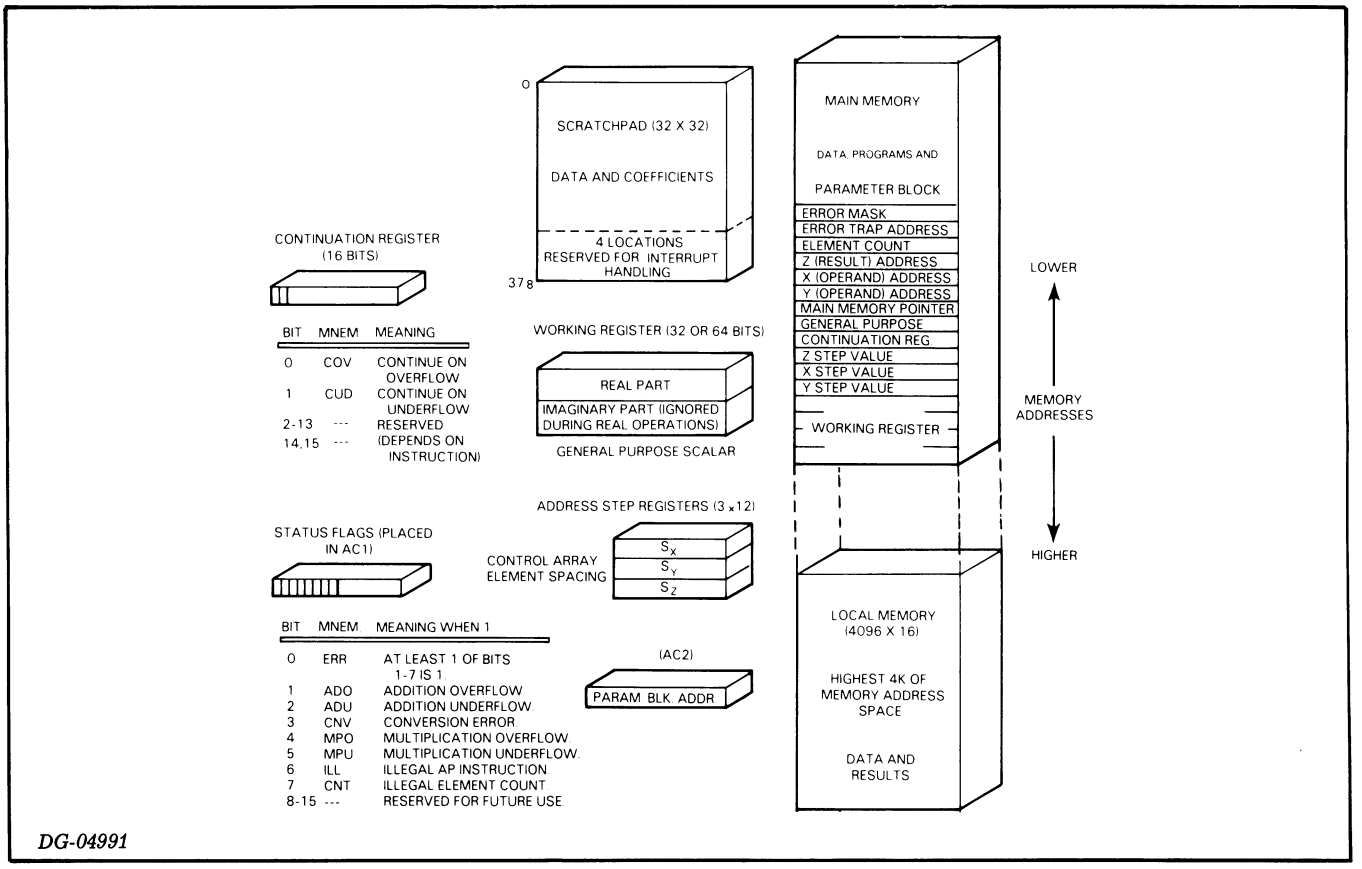
REAL ARRAY, STEP VALUE = +3

DG-04994


LOWER
MEMORY ADDRESSES
HIGHER
X_{n+6}
X_{n+4}
X_{n+2}
X_n
COMPLEX ARRAY, STEP VALUE = -2
$X_{n,n+1,n+2,..}$
REAL ARRAY, STEP VALUE = 0
DG-04993

Hardware Features

This section describes the registers and other components of the IAP which are accessible to the user program. The accompanying diagram shows a "programmer's model" or summary of the components and their relations to the CPU.



DG-04991

Local memory

The IAP has 4096 16-bit words of *local memory* to hold operands and results of calculations. The location of this memory in the CPU's physical address space is determined by jumper wires on the IAP board. These jumpers should be set so that the IAP local memory occupies the highest 4K of the memory address space. This memory may be mapped to any part of a user's logical address space by the ECLIPSE memory allocation and protection unit (MAP).

Non-IAP instructions may reference local memory in the same manner that they reference any other part of main memory. The IAP, however, interprets all indices as relative to the start of local memory. For instance, if you wished to reference the first scalar in local memory with an IAP instruction, you would use an index of 0 even though the scalar was located at, say, location 70000_8 of your logical address space.

Local memory addressing

The size of local memory is fixed at 4096 words. This places a restriction on the size of arrays which the IAP can manipulate. Any out-of-range references to local memory will be "wrapped around" by truncating to the appropriate number of bits: 11 for real numbers, or 10 for complex numbers.

Working register

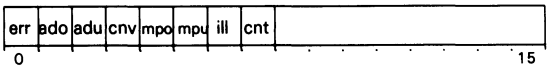
The IAP *working register* is a 64-bit general purpose register. It can be referenced by IAP instructions. Complex scalars occupy all 64 bits of the working register; real scalars occupy the high-order 32 bits.

Scratchpad memory

The IAP contains thirty-two 32-bit words of high speed *scratchpad* memory. This memory may be referenced by IAP instructions. It is also used to store the state of the CPU and IAP during an interrupt.

Status Flags

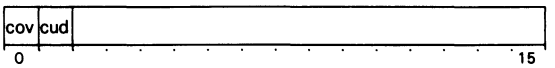
The IAP contains flags that indicate conditions such as overflow. The values of the status flags are placed in AC1 after an IAP instruction is executed. The IAP error handling logic allows the program to cause a trap to occur if any flag is set to 1. Unused flags are always set to 0. The following table describes the meanings of the status flags.



BITS	NAME	CONTENTS or FUNCTION
0	ERR	At least 1 of bits 1-7 is 1.
1	ADO	Addition overflow.
2	ADU	Addition underflow.
3	CNV	Conversion error: real scalar too large to be converted to an integer.
4	MPO	Multiplication overflow.
5	MPU	Multiplication underflow.
6	ILL	Illegal IAP instruction.
7	CNT	Illegal element count (zero or negative).
8-15	---	Reserved for future use.

Continuation register

The *continuation register* is used to control the IAP's response to various error conditions. Normally, if the IAP detects an error while processing an array, it terminates without any further processing. By setting bits in the continuation register, you may specify that the IAP is to continue processing after detecting an error. When an error occurs, if the appropriate bit is set to 1, the IAP generates a value to use as the result of the calculation, and then proceeds with execution of the instruction. The format of the continuation register, and the values which the IAP generates, are shown below.



BITS	NAME	CONTENTS or FUNCTION
0	COV	Continue after overflow error: use the greatest possible real number (approximately 16^{63}) as the result.
1	CUD	Continue after underflow error: use zero as the result.
2-15	---	Reserved (see below).

Some IAP instructions make use of other bits in the continuation register for various control functions. These functions are described in the sections on the specific instructions.

Interrupt handling

Many IAP instructions take a relatively long time to execute. For this reason, the IAP can be interrupted in the middle of an instruction, and resume operation at a later time with no loss of data. This enables IAP programs to run in a multi-user system without placing an excessive load on other users. (The interrupt facility does *not* enable the IAP itself to be time-shared between two users.)

If the CPU detects an interrupt while the IAP is running, the operating states of the CPU and IAP are stored into locations 34_8 - 37_8 of the scratchpad memory. Locations 34_8 and 35_8 save the states of several internal registers; locations 36_8 and 37_8 hold the contents of the working register. The working register itself is used to store several other internal registers.

When an interrupt occurs during an IAP instruction, the program counter points to the second word of the instruction. Upon returning from the interrupt, the CPU will execute this second word, which is interpreted by the IAP as a "reenter" command. All internal registers are restored from the working register and scratchpad, and processing resumes at the point where it was interrupted.

NOTE: *At the time when the interrupt reentry is performed, the working register and locations 34_8 - 37_8 of the scratchpad must contain the same data that they did after the IAP was interrupted. Undefined results will be produced if these data are not preserved.*

Interrupt timing

The maximum interrupt latency of an IAP system is 12us (assuming no data channel cycles). The maximum data channel latency is 4.5us.

Use of WCS

To use the IAP, you must initialize the writable control store (WCS) portion of the CPU's microprogram memory with the IAP firmware. This firmware is supplied to users as a file called APWCS.MO. Data General also supplies the microprogram loader, MICROLDR.SV, for use under RDOS.

NOTE: *The operation of the IAP is unpredictable unless WCS has been properly initialized.*

To initialize WCS for IAP use, type the following command to the RDOS Command Line Interpreter (CLI):

```
MICROLDR APWCS.MO
```

This command invokes the microprogram loader, causing it to load the contents of APWCS.MO into WCS. The IAP is then ready for use.

If you are not using RDOS to support your application, you can use the DTOS or DDOS WCS loader. DTOS and DDOS are small tape- and disc-oriented operating systems intended mainly for diagnostic use. To use the loader, mount the DTOS tape (or DDOS disc) on your system, and start it using the front panel switches. The specific procedure is as follows:

- Set the device code for the disc or tape drive in data switches 10-15 (standard device codes are 22_8 for tape, and 33_8 for disc).
- Set switch 0 to 1 (up), and switches 1-9 to 0 (down).
- Lift the **RESET** key.
- Lift the **PROGRAM LOAD** key to start loading DTOS.

The system will print a startup message on the console terminal, followed by an asterisk (*) which is your *prompt* to type a command. You type:

LOAD,APWCSxx ("xx" is the current revision number, which may change from time to time. Data General always supplies its customers with the most recent revision.)

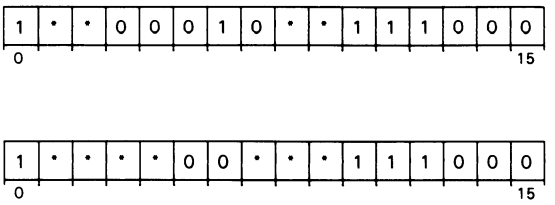
The WCS loader program will take control and print a message on the terminal to indicate that it is working. When it is finished, the system will type another asterisk. At this time you may halt the processor, remove the tape or disk, and proceed with your own application. The IAP microcode will remain in WCS as long as the processor is turned on.

For more information on using DTOS or DDOS, consult the DTOS User's Manual *(DGC no. 014-68).*

Programming

The IAP instructions use the **XOP1** function of the ECLIPSE S/250. **XOP1** is used in non-IAP systems to call subroutines, or to enter writeable control store on systems with that option. IAP instructions are two words (32 bits) in length. Each instruction has four bit fields which determine the specific operation to be performed. Thus a large number of different operations may be specified. Each operation has its own mnemonic which can be used instead of **XOP1** for assembler language programming. The format of the IAP instructions is shown in the following diagram.

AP INSTRUCTION FORMAT:



* Bits depend on instruction.

Parameter blocks

Many IAP instructions have a large number of options. They often require more arguments than can be fit into the instruction itself. For this reason all instructions use a parameter block consisting of up to 16 words of main memory. Your program must set up this block, and place the address of the first word of the block in AC2 before executing an IAP instruction.

The format of the parameter block is shown below. The functions given for each parameter word are typical; some IAP instructions use certain parameter words for functions other than those listed.

WORD	FUNCTION (typical)
0	Error mask.
1	Error trap address.
2	Element count.
3	Index of destination array.
4	Index of first source array.
5	Index of second source array.
6	Main memory pointer.
7	(Usage depends on instruction.)
8	Continuation register.
9	Destination step value.
10	First operand step value.
11	Second operand step value.
12-15	Scalar to be placed in IAP working register (32 or 64 bits).

No IAP instruction requires all 16 words of the parameter block as arguments. The IAP instruction set is organized so that, in general, each parameter word has a specific function. This organization simplifies programming, especially when several operations are to be performed on one array: a single parameter block may be used for several instructions, each of which uses different words from the block.

The IAP never modifies the contents of the parameter block. The only exception to this rule is when you specify an address that causes the IAP to overwrite the parameter block with result data.

Partial load instructions

Many IAP instructions support powerful options that may not always be useful. For this reason some instructions have a partial load option. The option causes the IAP to ignore certain parameter words. Generally this sacrifices some of the instruction's power, but results in an increase in speed.

In assembler language, you may specify a partial load instruction by appending a P to the regular instruction mnemonic. The particular parameter words to be used or ignored on a partial load depend on what instruction is being executed.

One of the options which is suppressed by a partial parameter load is the loading of new values into the address step registers. This does not mean that the step values are ignored; it simply means that all arrays will be stepped using the current values of the step registers, i.e., whatever was placed in them by a previous IAP instruction.

Error handling

Two words of the parameter block control the IAP error trapping facility. Word 0 of the block is an error mask, which is logically ANDed with the status flags at the completion of each IAP instruction. The bit assignments for the error mask are identical to those described previously in the section on "Status flags."

If the result of the logical AND is nonzero, a trap occurs. Thus the error mask allows you to select which types of errors will produce a trap.

When a trap occurs, the contents of the accumulators, carry bit, and program counter are pushed onto the stack. Then the program jumps to your error handling routine, the address of which is taken from word 1 of the parameter block. This address may be indirect.

The error handling routine may exit with a *Pop block* instruction, which will restore the PC, carry bit and accumulators from the stack. Program execution will resume with the instruction following the IAP instruction that caused the trap. The accompanying diagram shows the action of the IAP error handling facility.

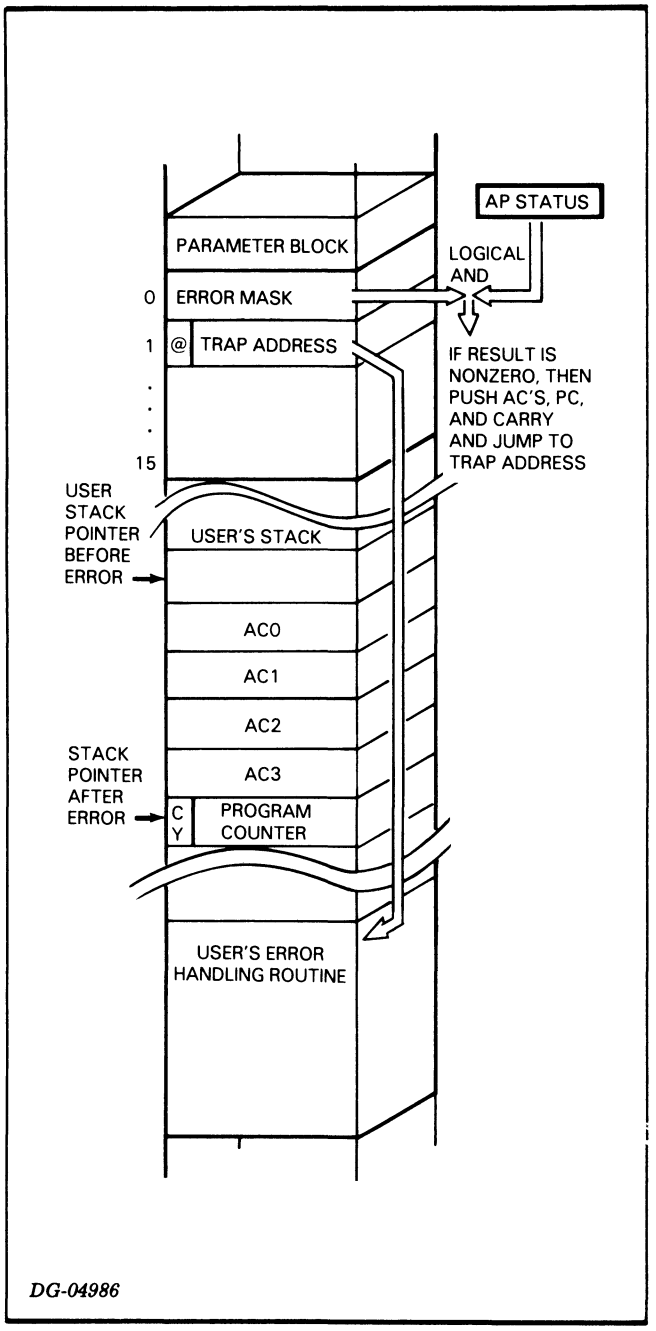
NOTES: *All addresses are in your logical address space; therefore you must provide a stack and an error handling routine in order to use the IAP trapping facility.*

Setting bits in the continuation register does not disable the error trapping facility; it merely causes the trap to be delayed until the instruction completes all calculations. Normally the trap occurs as soon as the error condition is detected.

The error mask and trap address in the parameter block are always active, regardless of whether a full or partial parameter load is specified.

Accumulator usage

You must reserve several accumulators for use by the IAP. As mentioned above, AC1 is used by the IAP for returning error status. In general, AC0 and AC1 are *always* assumed to be available to the AP; therefore you should not have any critical data in AC0 or AC1 when excecuting an IAP instruction. AC2 is sometimes used by the AP, but it is saved and restored using an internal register, so the only time its contents might seem to have changed is during an interrupt. AC3 is always preserved by the IAP.


AP STATUS
PARAMETER BLOCK
LOGICAL AND
0
ERROR MASK
1
@
TRAP ADDRESS
IF RESULT IS NONZERO, THEN PUSH AC'S, PC, AND CARRY AND JUMP TO TRAP ADDRESS
15
USER STACK POINTER BEFORE ERROR
USER'S STACK
AC0
AC1
AC2
STACK POINTER AFTER ERROR
AC3
C
Y
PROGRAM COUNTER
USER'S ERROR HANDLING ROUTINE
DG-04986

APPENDIX C
STANDARD I/O DEVICE CODES

OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME	OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
00	----	--	Unused	41[3]	DPO	8	IPB full duplex output
01	----	--	Unused	40	SCR	8	Synch. communication receiver
02	ERCC	--	Error checking and correction	41	SCT	8	Synch. communication transmitter
03	MAP	--	Memory allocation and protection unit	42	DIO	7	Digital I/O
04[5]	ISPC	5	Host, SP MAP	43	DIOT	6	Digital I/O timer
					PIT	6	Programmable Interval Timer
05				44	MXM	12	Modem control for MX1/MX2
06	MCAT	12	Multiprocessor adapter transmitter	45			
07	MCAR	12	Multiprocessor adapter receiver	46	MCAT1	12	Second multiprocessor transmitter
10	TTI	14	TTY input	47	MCAR1	12	Second multiprocessor receiver
11	TTO	15	TTY output	50	TTI1	14	Second TTY input
12	PTR	11	Paper tape reader	51	TTO1	15	Second TTY output
13	PTP	13	Paper tape punch	52	PTR1	11	Second paper tape reader
14	RTC	13	Real-time clock	53	PTP1	13	Second paper tape punch
15	PLT	12	Incremental plotter	54	RTC1	13	Second real-time clock
16	CDR	10	Card reader	55	PLT1	12	Second incremental plotter
17	LPT	12	Line printer	56	CDR1	10	Second card reader
20	DSK	9	Fixed head disc	57	LPT1	12	Second line printer
21	ADCV	8	A/D converter	60	DSK1 or SPO	9	Second fixed head disc or SP
22	MTA	10	Magnetic tape	61	ADCV1 or SP1	8	Second A/D converter or SP
23	DACV	--	D/A converter	62	MTA1 or SP2	10	Second magnetic tape or SP
24	DCM	0	Data communications multiplexor	63	DACV1 or SP3	--	Second D/A converter or SP
25				64	SP4		SP
26	DKB	9	Fixed head DG/Disc	65	SP5		SP
27	DPF	7	DG/Disc storage subsystem	66	DKB1 or SP6	9	Second Fixed Head DG/Disc or SP
30	QTY	14	Asynch. hardware multiplexor	67	DPF1 or SP7	7	Second DG/Disc storage subsystem or SP
30	SLA	14	Synchronous line adapter	70	QTY1	14	Second asynch. hardware mux
31[1]	IBM1	13	IBM 360/370 interface	70	SLA1	14	Second synchronous line adapter
32	IBM2	13	IBM 360/370 interface	71[1]		13	Second IBM 360/370 interface
33	DKP	7	Moving head disc	72		13	Second IBM 360/370 interface
34[1]	CAS[1]	10	Cassette tape	73	DKP1	7	Second moving head disc
	DCU[4]	4	Data Control Unit				
34	MX1	11	Multiline asynchronous controller	74	CAS1	10	Second cassette tape
35	MX2	11	Multiline asynchronous controller	74[1]		11	Second multiline asynch. controller
36	IPB	6	Interprocessor bus--half duplex	75		11	Second multiline asynch. controller
37	IVT	6	IPB watchdog timer	76	DPU	4	DCU To Host Interface
40[2]	DPI	8	IPB full duplex input	77	CPU	--	CPU and console functions

DG-07317

1. Code returned by INTA and used by VCT
2. Can be set up with any unused even device code equal to 40 or above
3. Can be set up with any unused odd device code equal to 41 or above
4. Can be set to any unused device code between 1 and 76
5. Only valid for SP.

APPENDIX D
OCTAL AND HEXADECIMAL CONVERSION

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

To convert a decimal number to octal or hexadecimal:

1. Locate the largest decimal value in the appropriate table that will fit into the decimal number to be converted;

2. Note its octal or hex equivalent and column position;

3. Find the decimal remainder.

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

OCTAL CONVERSION TABLE

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

HEXADECIMAL CONVERSION TABLE

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

APPENDIX E
ASCII CHARACTER CODES

APPENDIX F
BINARY, OCTAL AND DECIMAL NUMBERING SYSTEMS

The most familiar numbering system in our society is the decimal system. For ordinary mental or pencil-and-paper work it is clearly the easiest to use. Computers, however, use the binary system, which becomes very confusing to humans when more than a few digits are involved. Fortunately, binary can be easily translated into octal or hexadecimal representation, both of which are relatively easy for humans to use.

In this section, we provide some basic background on the binary, octal and hexadecimal numbering systems. Most readers will already be familiar with these, but some may not and others may find the review helpful.

The binary numbering system is used in computers because the two binary values can be easily represented electronically. In the binary system, the only two permissible digits are 0 or 1, and each position in a binary number represents some power of 2. For example, consider the binary number:

1011010_2

which is equivalent to the sum (in decimal):

$(1x2^6) + (0x2^5) + (1x2^4) + (1x2^3) + (0x2^2) + (1x2^1) + (0x2^0)$

or

$64 + 0 + 16 + 8 + 0 + 2 + 0 = 90_{10}$.

If we divide this number into groups of 3 starting at the right, thus:

1 011 010,

we see that each group of 3 has a range of:

000 = 0

to

$111 = 7 = (2^2 + 2^1 + 2^0) = (4 + 2 + 1)$.

Zero to 7 is the range of digits allowable in the octal numbering sytem, so we can convert from binary to octal simply by grouping and evaluating each group of 3 binary digits by itself. In octal, the number above becomes:

1 011 010

or

$1\ 3\ 2 = 132_8$

We can also convert this number to hexadecimal (or base 16). Zero through nine *decimal* are unchanged in the hexadecimal system, but $10\text{-}15_{10}$ are represented by the letters A through F.

If we divide the original binary number into groups of 4 instead of 3, starting from the right, we get:

101 1010

The range for one group is now:

0000 = 0

to

$1111 = 2^3 + 2^2 + 2^1 + 2^0$
$= (8 + 4 + 2 + 1) = 15_{10} = F_{16}$

The number in the example above is then:

101 1010

or

$5\ A = 5A_{16}$

This page intentionally left blank.

APPENDIX G
ECLIPSE/NOVA LINE COMPATIBILITY

The ECLIPSE S250 is compatible with Data General's NOVA line. You may execute any NOVA-based program on an ECLIPSE series computer, with the following restrictions:

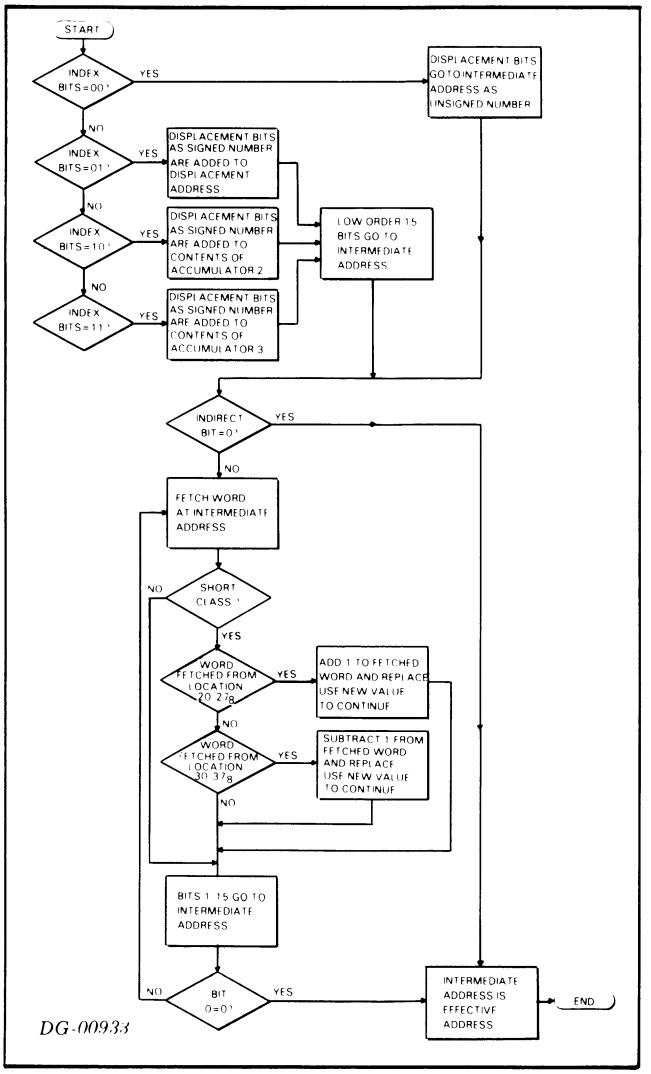
- Make sure your NOVA-based programs are not dependent on instruction execution times or I/O transfer times. ECLIPSE times may be faster.
- Make sure that your ECLIPSE system is equipped with at least the same amount of memory and number of I/O devices as the NOVA system on which you executed your programs.
- Make sure that your NOVA-based programs do not use any of the following:
    - ALC instructions specifying both the *no load* and the *never skip* options (ECLIPSE processors recognize this bit combination in other standard instructions);
    - the NOVA MAP;
    - the data channel increment or add-to-memory features.

Other notes:

- The NOVA hardware multiply/divide option and the standard ECLIPSE multiply/divide function the same way. Note, however, that the NOVA multiply and divide operation codes are not the same as the ECLIPSE operation codes.
- The NOVA floating point and the ECLIPSE floating point option use similar (not identical) instructions, but do not function the same way. Make sure you use the correct instruction format.
- Floating point data formats are the same for both NOVA and ECLIPSE processors.

APPENDIX H
ADDRESSING

A flow diagram of the addressing process is shown below. See Chapter III for a detailed discussion of addressing.



APPENDIX I
BOOTSTRAP LOADER

The *Program Load* console switch loads the bootstrap loader program shown below into the first 32_{10} words of memory and starts the program at location 0. See the console section of Chapter II for details on the use of the Program Load function.

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

INSTRUCTION INDEX

I/O INSTRUCTION INDEX

INTEGRAL ARRAY PROCESSOR INSTRUCTIONS

BIBLIOGRAPHY

The following Data General publications may be of interest to readers of this manual:

Programmer's Reference, Peripherals	DGC No. 015-000021
Programmer's Reference, Data Control Unit	DGC No. 015-000060
Technical Reference, Data General Communications System	DGC No. 014-000070
Technical Manual, 6020 Series Tape Transport	DGC No. 015-000040
Technical Manual, Model 6045 6050 6051 Disc Drive (10 Megabyte)	DGC No. 015-000057
Technical Manual, DG/Disc Storage Subsystem (6060 Series, 100 Megabyte)	DGC No. 015-000061
Technical Manual, Model 6063-6065 Fixed Head Disc	DGC No. 015-000072
Interface Designer's Reference, NOVA and ECLIPSE Line Computers	DGC No. 015-000031
Software Summary and Bibliography	DGC No. 093-000110
AOS Software Documentation Guide	DGC No. 093-000202
AOS Programmer's Manual	DGC No. 093-000120
AOS Macroassembler Reference Manual	PGC No. 093-000192
AOS Binder User's Manual	DGC No. 093-000190
AOS Debugger and Disk File Editor User's Manual	DGC No. 093-000195
AOS System Manager's Guide	DGC No. 093-000193

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

DG-04976

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

DataGeneral

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

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

Postage will be paid by addressee:

Data General

ATTN: ENGINEERING PUBLICATIONS
4400 Computer Drive
Westboro, MA 01580

CUT ALONG DOTTED LINE

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

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
C/350, C/330, C/300		
S/250, S/230, S/200		
S/130		
AP/130		
CS Series		
N3/D		
Other NOVA		
microNOVA		
Other (Specify) _____		
_____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ SOS
- ☐ RDOS
- ☐ RTOS
- ☐ Other

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ ECLIPSE Cobol
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Interactive
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ RSTCP
- ☐ HASP
- ☐ RJE80
- ☐ SAM
- ☐ CAM
- ☐ 4025
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

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

○ _____

DG-05810

Data General

Data General Corporation, Westboro, Massachusetts 01581, (617) 366-8911

FOLD FOLD

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

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

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator
4400 Computer Drive
Westboro, MA 01580

Data General

Data General Corporation, Westboro, Massachusetts 01580