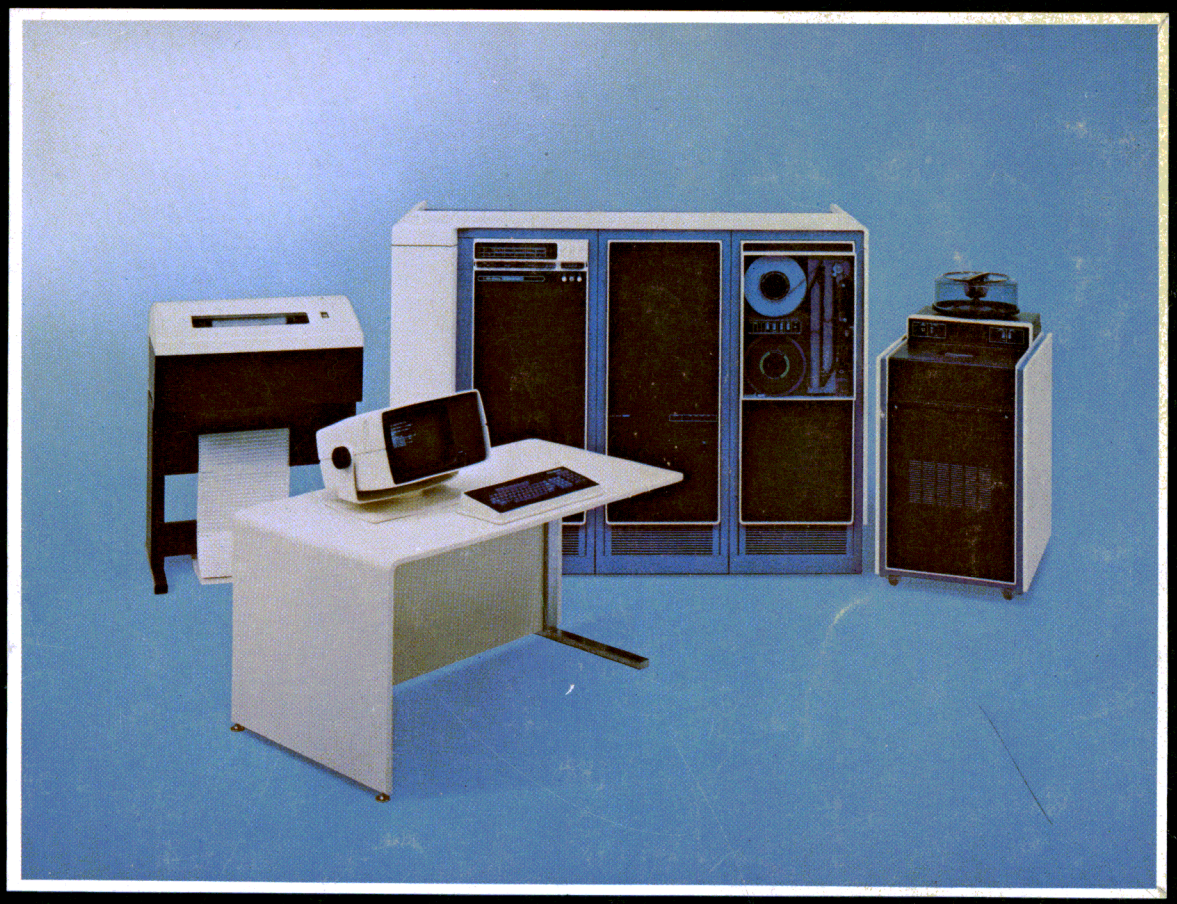
ECLIPSE® M/600
PRINCIPLES OF OPERATION



Data General

Data General Corporation, Westboro, Massachusetts 01581

ECLIPSE® M/600
PRINCIPLES OF OPERATION

DataGeneral
Data General Corporation, Westboro, Massachusetts 01581

NOTICE

FIRST EDITION

(First Printing, January 1978)

SECOND EDITION

(First Printing, April 1978)

Ordering No. 014-000092

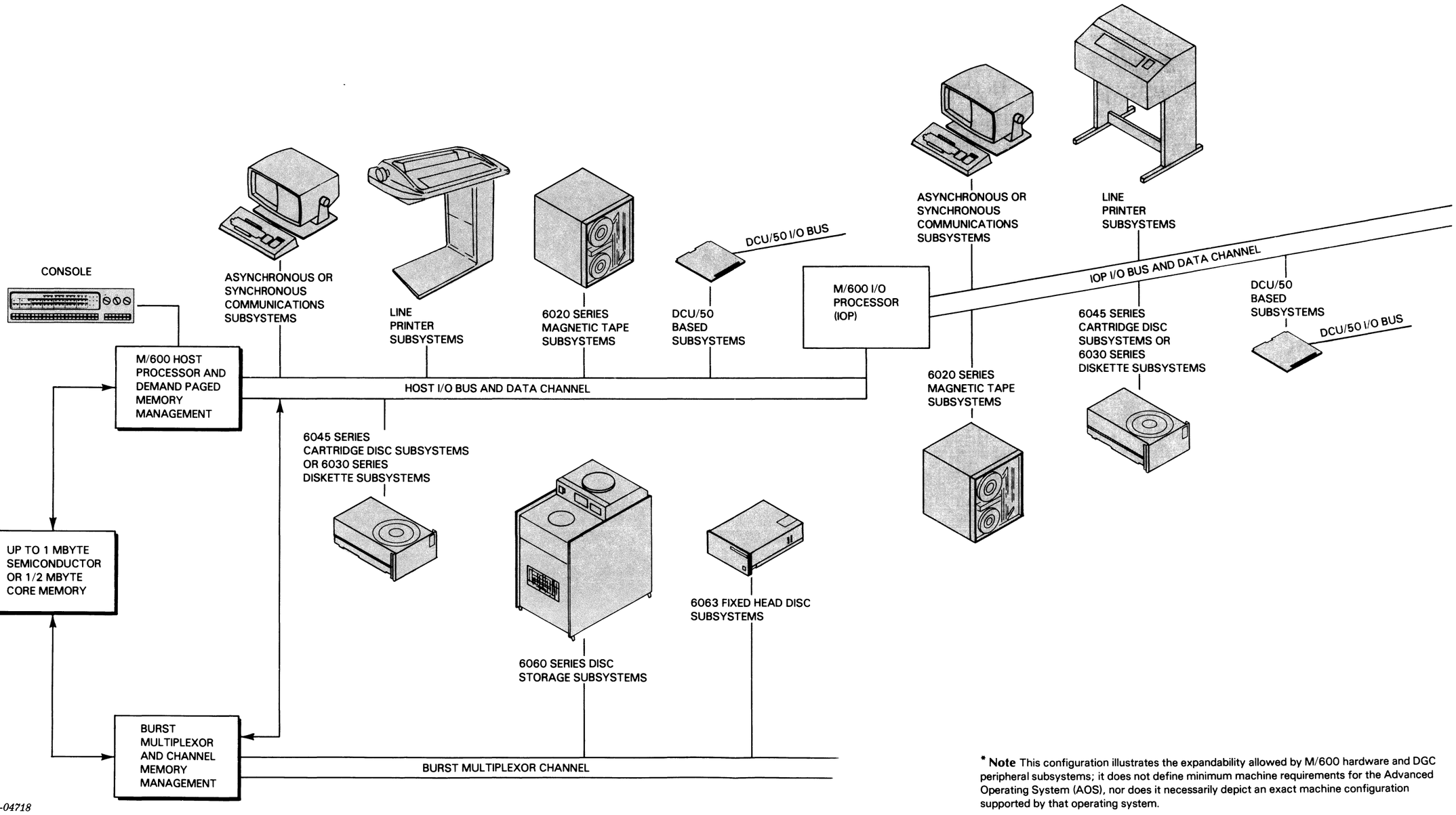
Printed in the United States of America
Rev. 01, April 1978

CONTENTS

CONTENTS CONTINUED

This page intentionally left blank.

ECLIPSE M/600 CONFIGURATION*



DG-04718

*** Note** This configuration illustrates the expandability allowed by M/600 hardware and DGC peripheral subsystems; it does not define minimum machine requirements for the Advanced Operating System (AOS), nor does it necessarily depict an exact machine configuration supported by that operating system.

Chapter I
INTRODUCTION TO M/600

Data General's ECLIPSE M/600 computers provide exceptional throughput and response characteristics in Multi-user system environments. M/600 computer systems are a balanced blend of technical features such as Advanced Input/Output Management, demand paged memory management, high processing speed, and powerful software control. The M/600 computer marks Data General's leadership in online, medium-scale, reliable, computers.

The M/600 combines advancements in hardware specifically designed to perform in a multiprogramming operating system environment. The sophisticated three-line Input/Output Management System helps clear the I/O bottlenecks which often limit system performance. A demand-paged one-megabyte main memory facility optimizes memory use in multiuser applications and extends effective memory space. M/600's main frame type packaging ensures high reliability and maintainability. Software support by Data General's Advanced Operating System (AOS) makes this hardware performance available to up to 64 users for concurrent development and execution of timesharing, batch and real-time programs.

Input/Output Management System (IOMS)

The M/600 Input/Output Management System consists of a three-level hierarchy, including a high speed Burst Multiplexor Channel (BMC), a standard Data Channel, and an independent Input/Output Processor (IOP).

The BMC is a high speed direct communications pathway between main memory and high performance peripherals like Fixed Head DG/Disc Subsystems and 96- or 190-megabyte DG/Disc Storage Subsystems. Data Transfer rates on the BMC can attain the full main memory system bandwidth, resulting in a very fast aggregate rate - 10 megabytes per second. Up to eight high speed controllers can connect to the M/600 BMC, resulting in a potential system expandability of up to six billion bytes.

A Standard ECLIPSE Data Channel comprises the second level of the M/600's I/O control hierarchy. It handles the communications between the systems Job Processor and medium-speed peripheral equipment such as magnetic tape and cartridge disc drives. It also acts as a high-speed -- up to 2.5 megabytes/second -- interface between the Job Processor and the third-level Input/Output Processor (IOP) for low-speed peripheral devices.

The IOP is an independent processor with an ECLIPSE instruction set and 64K bytes of local memory. It processes and buffers data input from low-speed peripherals such as DASHER display and printer terminals, card readers and asynchronous communications devices. The IOP acts as a front-end processor for the M/600 system, off loading the Job Processor by buffering all low-speed input, checking it for errors and sending it to the Job Processor one logical record at a time. The IOP also receives all terminal output from the Job Processor and transmits the data back to as many as 64 terminals at the speeds they require. By absorbing the character-handling overhead in large multiterminal applications, the IOP frees up the Job Processor for high-speed processing.

Intelligent Memory Management

In M/600 systems, an efficient Demand-Paged Memory Management Facility under the control of the Advanced Operating System (AOS) executive optimizes multiple user access to main memory by reducing each user's requirement at any given time. This is done by automatically dividing each user's memory space into "pages", and keeping only active pages in main memory, while other pages are stored on fast access devices like fixed head discs.

The Demand-Paged Memory Management Facility, working with an AOS page replacement algorithm in the executive, is a predictive memory management tool transparent to system users. The AOS software analyzes and predicts what pages will be needed at any given time by any given user, and the system hardware moves the pages into main memory. The adaptive nature of the software is possible through analysis of system information about memory use. The hardware maintains information on page validity and page usage, as well as the type of information on these pages (procedures or data). The executive uses this information, together with user priority data and other resource requirements for each user to determine the best overall mix of users at any given time.

M/600 systems are available with MOS memory, expandable to one megabyte, and core, expandable to 512K-bytes. MOS storage has read/write cycles of 500- and 700 nanoseconds, respectively, while the core storage has cycle times of 800 nanoseconds. Core can be expanded in 32K-byte increments to achieve the maximum storage, and the semiconductor memory can be expanded in 64K-byte increments. All semiconductor memory has error checking and correction (ERCC) to maintain maximum data integrity.

Fast Job Processor

The heart of the M/600 system is a fast, 200-nanosecond microcycle Job Processor with an advanced microprogrammed architecture that incorporates the standard Data General ECLIPSE instruction set plus extensions. Instructions include both 16-bit single-word instructions for efficient memory utilization, as well as 32-bit double-word instructions -- for extended addressing range. Word and block move instructions transfer large amounts of data with one instruction. Absolute, relative, indexed, immediate and indirect addressing are available in both single- and double-word formats.

M/600 system instruction sets include four source data processing extensions for both business and scientific applications, beyond the standard ECLIPSE instruction set.

The business source data processing extensions include a character instruction set and decimal/edit instruction set with console functions to aid programmers in tracing and debugging programs. Scientific extensions to the standard ECLIPSE instruction set have been designed to handle floating-point functions like exponentation, logarithms and sines. M/600 systems perform 64-bit floating point operations at ultra-high speed: addition takes 0.8 microseconds, and division takes 6.8 microseconds.

The M/600 instruction set is upward-compatible with the standard NOVA°°° and ECLIPSE instruction sets.

Advanced Packaging

The M/600 system design incorporates conservative operating margins in its power, cooling and logic subsystems to help electrical parts last longer, minimize the possibility of transient errors, lower the system's vulnerability to component aging, and allow field interchangeability of parts for quick maintenance.

In addition, M/600's high-density logic technology keeps mechanical interconnection to a minimum; multilayer circuit board technology helps to keep signals "clean"; and a high-capacity power supply and distribution system provides the power levels necessary for M/600's high performance logic. Printed circuit boards are installed vertically to ensure good air flow through the chassis. A heavy-duty air cooling system keeps high power components cool and prevents overheating to help guard against permanent damage. The M/600 cabling scheme provides maximum stress relief and easy accessability to all connections.

Diagnostics

The M/600 system elements perform several types of diagnostic functions to keep system uptime high. Error control features include error checking and correction (ERCC) in the main semiconductor storage and high-speed discs.

Parity error control is used in the IOP memory, the BMC, and the Job Processor control store to trap errors as they occur and help pinpoint problems.

M/600 systems automatically execute a self-test routine on system initialization to verify correct operation of the Job Processor and Main Storage. On-line diagnostics constantly run as a task under AOS in M/600 systems and include an ongoing check of processor logic paths, memory and major peripherals. On-line diagnostics also permit major peripherals to be taken off-line for maintenance, then verified for correct operation on-line without shutting down normal system operation.

Communications

The M/600 system can support a wide variety of communications systems, using existing Data General communications equipment, including:

- Half- or full-duplex asynchronous communications at speeds up to 19.2 kilobaud;
- Transparent and non-transparent synchronous communications at speeds up to 56 kilobaud, using line modules compatible with IBM's binary synchronous communications protocol (BISYNCH).

M/600 systems can communicate with all other Data General systems, and IBM-compatible computers through Data General's RJE80 and HASP II Workstation Emulator software packages.

RJE80 supports data transfer in point-to-point or multidrop modes among networks of Data General computers acting as master or slave stations. The HASP II Workstation Emulator allows communications with other Data General computer systems and most other mainframes using sophisticated multileaving and data compression techniques. Both RJE80 and HASP are supported by synchronous line handlers that support high throughput communications processing while combining a range of multiplexing capabilities.

In addition, up to 15 M/600's or other Data General NOVA and ECLIPSE systems may be interconnected by a Multiprocessor Communications Adapter to distribute large processing loads.

This page intentionally left blank.

Chapter II
M/600 COMPUTER FEATURES

MEMORY ALLOCATION AND PROTECTION

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 M/600 MAP unit provides the hardware necessary to control and use more than 32K words 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 1,048,576 words (1M) 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 device used by a particular user or 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 M/600 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 M/600. *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 M/600 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 at any time 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 at any time 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 at any time.

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* instruction has the same bit pattern as some of the I/O instructions. The MAP therefore has a *Lef* mode bit, which switches the mode of the M/600 from *Lef* mode to I/O mode. When the *Lef* mode bit is 1, 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.

DEMAND PAGING

Introduction

You can significantly reduce the system overhead of a time-sharing operating system, under certain circumstances, through the use of a demand paging memory control algorithm. In this section, we will discuss the M/600 demand paging facility and the circumstances in which it can be most useful. A detailed discussion of demand paging concepts is beyond the scope of this manual, but many texts are available on this subject.

In a non-paged time-sharing system, your entire logical address space (up to 65,536 bytes) must be represented in physical (main) memory whenever your program is being serviced. In most programs, however, active processing tends to concentrate in one area of the program. As execution progresses, this area of concentrated processing will slowly migrate to other parts of the program, but only rarely will large areas of the program be involved at once. As a result, memory in a non-paged time-sharing system usually contains many pages of user address space that are not actually needed in the present processing.

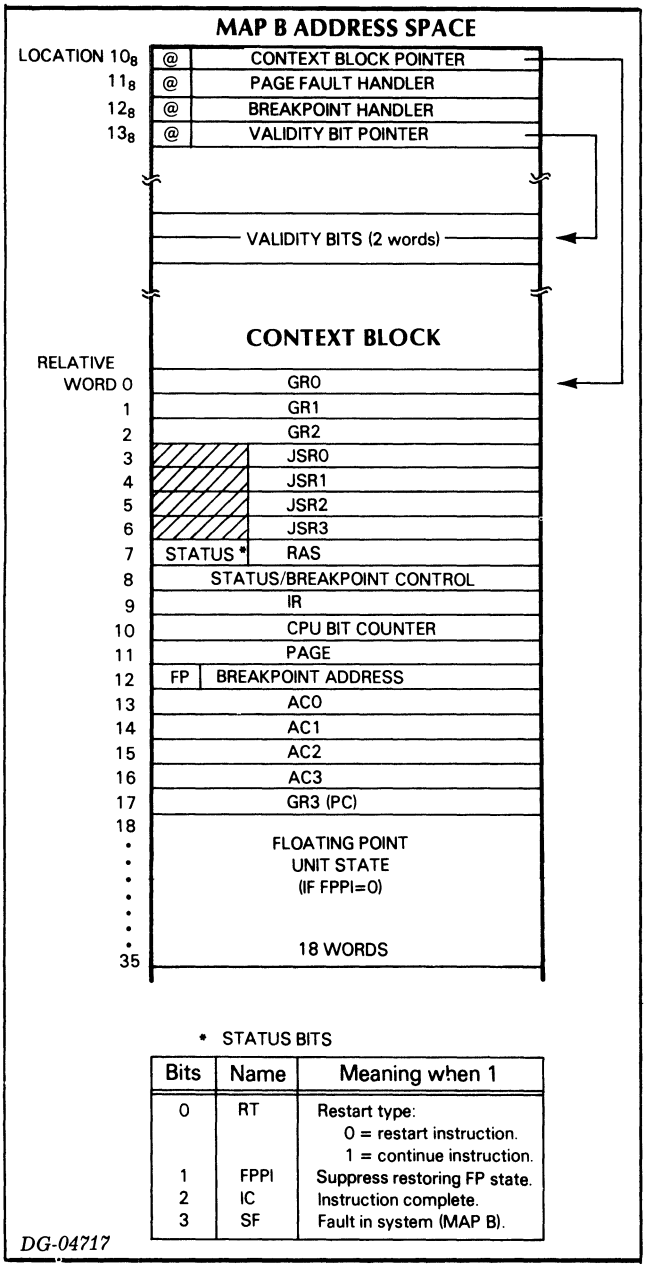
The M/600 demand paging facility, on the other hand, can work with individual areas (called *pages*) of 2048 bytes each, bringing into main memory only those pages needed for the present processing. Page-swapping will be necessary only occasionally, and even then, the system need only swap one or two pages instead of swapping an entire address space at once. The obvious result of all this is significantly reduced system overhead.

Page Faults

In a demand paged system, many of the 32 possible pages of each user's logical address space may not be represented in main memory at any moment. Some of these pages may not be part of the active program at all and thus be declared *invalid* by the map (see the section on the MAP in this chapter). Other pages may contain parts of the active program but be *swapped out* and declared invalid by the demand paging facility.

In either case, referencing a location on such a page (e.g., by branching to that location or referencing data in that location) will cause a fault because you are attempting to reference a location that is not in memory.

When you attempt to reference a location which is not part of your allocated user space at all (i.e., a location which has been declared invalid by the map), a *validity fault* occurs. Control is transferred to the protection fault handler, but usually a validity fault is a fatal error and no more processing of your program is possible. See the MAP section in this chapter for a discussion of validity faults.



When you attempt to reference a location which *is* part of your user space but which is not in memory at the moment (i.e., a location which has been declared invalid by the demand paging facility), a *page fault* occurs. With demand paging, this need not be a fatal error since the demand paging facility can freeze processing until the required location can be retrieved from a disc, and then restart the program where the fault occurred.

The M/600 demand paging facility does this by storing the state of the CPU, retrieving the page containing the required location, and then continuing (or in some cases, restarting) the interrupted instruction. All this occurs without visible effect on the program.

Validity Bits

Each page has a *validity bit* associated with it. Referencing a page whose validity bit has been set to 1 forces a validity error (which initiates a *map protection fault*). Referencing a page whose validity bit has been set to 0 forces a page fault sequence. The supervisor program has the responsibility to maintain the proper value of the validity bit for each page of user space.

The Validity Bit Pointer (location 13_8 of **MAP B** address space) points to two consecutive memory words containing the validity bits for the address space of the user presently being serviced.

Context Blocks

It is possible for a page fault to occur in the middle of an instruction (e.g., a **LDA** instruction which accesses a location on a page not presently in memory). Therefore, the demand paging facility must store the state of the CPU in a form which will permit continuing (or, in some cases, restarting) the instruction with no visible effect on program execution.

The demand paging facility does this by saving an 18-word *context block* which stores the contents of all the internal registers needed to define the state of the CPU. Location 10_8 in **MAP B** contains a (possibly indirect) pointer to the start of this context block. If the page fault occurred in **MAP A** address space, the demand paging facility also stores the state of the floating point unit in an additional 18 words immediately after the context block.

After all required values have been stored, the demand paging facility transfers control to the page fault handler via the page fault handler address (which may be indirect) in location 11_8 of **MAP B** address space. The page fault handler is part of the (software) operating system.

The page fault handler must retrieve the page that caused the fault by setting up an appropriate data channel transfer from the disc or other long-term storage device. Once the page has been brought into main memory, the page fault handler can restart the program by executing a **DPOP** instruction which restores the CPU state from the information in the context block.

Page-Use Flags

The hardware maintains a table of two flags for each page in user address space. One flag is set to 1 when any word on the page is *referenced*. The other is set to 1 when any word on the page is *modified*. Since four words (64 flags) are required for a 32K address space, and separate tables are maintained for the **Map A** and **Map B** address spaces, the demand paging facility maintains a total of 8 words. You can read or clear the page-use flags one word at a time using I/O instructions.

Since parts of a large program may be swapped in and out hundreds of times during a typical run, the paging algorithms should be as efficient as possible. The page-use flags may be used to improve the efficiency of paging software. There are many books available on this subject; here we will limit ourselves to a discussion of the information contained in the page-use flags and a very simple discussion of their uses.

The *reference* bits indicate which pages of a program have been used since the reference bits were last read and set to zero. You can detect little-used pages by regular examination of these bits while the program is running. Such pages can be swapped out to make room for more heavily used pages.

The *modified* bits indicate which pages have been modified since the modified bits were last read and set to zero. The software can often save time by checking the modified bits before retrieving a recently used page from memory. If the page has not been modified, the copy of the page on the swapping device is still valid.

Breakpoint Facility

The M/600 breakpoint facility permits trapping certain CPU actions to assist program development and debugging. This facility can ease the job of finding certain types of program bugs, notably those in which some memory location is spuriously overwritten by an unknown part of a program. The facility can also be used to monitor program performance.

Three types of traps are available: a *procedure trace* trap, an *address* trap, and a *write* trap. In all three cases, a trap turns on **Map B**, saves a context block (as described for page fault handling), and transfers control to the address of a breakpoint handler which is pointed to by location 12_8 of **MAP B** address space (the pointer may be indirect).

You can use the procedure trace trap to monitor the actions of any routines which employ the **SAVE** or **RETURN** instructions. If trace is enabled, executing of either of these instructions causes a trap. You may determine the type and location of the instruction by examining the context block.

You can use the address trap to interrupt program execution when any reference is made to the logical address contained in the Breakpoint Address Register. The write trap is similar to the address trap, but the hardware will only initiate a fault sequence if the specified memory location is modified.

> **NOTE:** *To use the breakpoint facility, demand paging must be enabled. If you wish to use the breakpoints in a non-paging system, you can effectively inhibit the paging facility by setting all 32 validity flags (pointed to by location 13_8 in* **MAP B** *) to 1.*

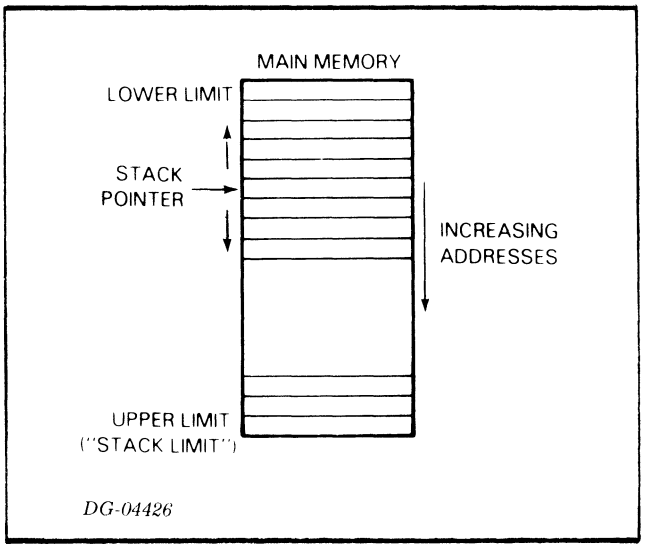
THE STACK

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 M/600 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, the program counter or the frame pointer (see below), and the carry bit in bit 0 of the last word pushed. The *Vector* instruction can put onto the stack a combination of a return block and individual words totalling up to 10 words, and the floating point instruction set uses a return block of 18 words.

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, and 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 the stack pointer. The updated value of the stack pointer is always stored in location 40_8.

The lower limit of the stack is determined by the initial value of the stack pointer, which is placed in location 40_8 when the stack is set up by the program. The upper limit is controlled by the value in location 42_8. This value is also chosen when the stack is set up, but it can be changed by the program if more stack area becomes necessary. Two other reserved locations are used to control the stack. Location 43_8 contains the address of the Stack Fault routine. Control is transferred to the Stack Fault routine when a stack underflow or overflow occurs (see Stack Protection, below). Location 41_8 contains the current value of the frame pointer, which is used as a reference pointer in the stack.

Stack Control Words

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

Stack Pointer

The stack pointer is the address of the current top of the stack. Its current value is always in location 40_8. A push operation increments the stack pointer by 1 and places the pushed word in the location addressed by the new value of the stack pointer. A pop operation takes the word addressed by the current value of the stack pointer, places it in a register and decrements the stack pointer by 1.

The value of the stack pointer when the stack is set up determines the lower limit of the stack.

Stack Limit

The stack limit is the upper limit of the stack area. After each push operation, the stack pointer is compared with the stack limit. If the stack pointer is greater than the stack limit, an overflow condition exists. The stack limit is contained in location 42_8.

Stack Fault Address

If a stack overflow or underflow occurs, control is transferred to the Stack Fault routine. The address of this routine, which may be indirect, is contained in location 43_8.

Frame Pointer

The frame pointer differs from the stack pointer in that it is not changed by push or pop operations, and so its value is not incremented or decremented. This makes it a useful reference pointer when it is set to the same value as the stack pointer, because it then preserves the original value of the stack pointer.

The frame pointer is used by the *Save* and *Return* instructions to store and reset the value of the stack pointer when entering or leaving subroutines. The frame pointer can also be used to define the boundary between words placed in the stack by a calling routine and words placed by a called routine. Using the frame pointer as a reference, a routine can go back into the stack and retrieve variables left there by the preceding procedure.

The frame pointer is contained in location 41_8.

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 will be pushed into areas that are reserved for other purposes, possibly overwriting 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 77777_8.

To be meaningful, the stack limit must be 10 to 23 addresses lower than the last word in the stack, because stack overflow is detected only at the end of a push operation (except in the case of the *Save* instruction - see details in the discussion of the *Save* instruction below). 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

After every operation that pushes data onto the stack, a check is made for overflow. 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.

Determine the nature of the fault by looking at bits 1-15 of the stack pointer and the stack limit. There are three possibilities:

- If the address contained in the stack pointer is not greater than the address in the stack limit, then the error was a stack overflow error resulting from the execution of a *Save* instruction.
- If the address in the stack pointer is greater than the address in the stack limit by a value greater than 5, then the error was a stack overflow error.
- If the address in the stack pointer is greater than the address in the stack limit by exactly 5, then the error was a stack underflow error.

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.
- 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

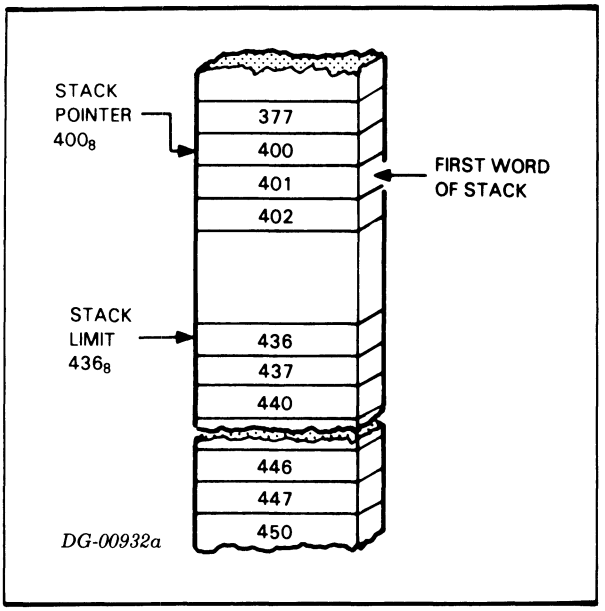
Initialize the stack fault address to the address of the routine that is to receive control in the event of a stack overflow or underflow. Bit 0 may be set to 1 to indicate an indirect address.

Frame Pointer

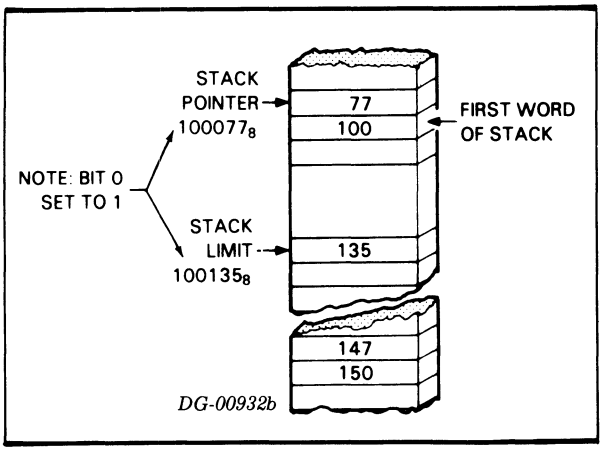
If the main user program is going to use the frame pointer, initialize it to the same value as the stack pointer. Otherwise, the frame pointer can be initialized in a subroutine by the *Save* instruction.

Examples

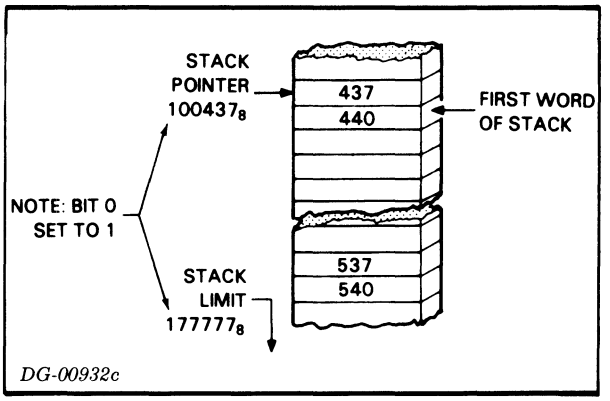
Stack area 50_8 words with underflow protection:



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



Stack area 100_8 words, no protection:



The first of the preceding stack arrangements could be set up using the following assembly language instructions:

```
        .TITL   STACK

        .EXTN   STH             ;Declare STH external

        .LOC    401             ;Go to location 401
        .BLK    50              ;Allocate 50 (octal) words

        .LOC    40              ;Go to stack control words
        400                     ;Stack pointer
        400                     ;Frame pointer
        436                     ;Stack limit
        @SFT                    ;Stack fault address

SFT:    STH                     ;Address of stack handler

        .END
```

INPUT/OUTPUT

This section describes Input/Output (I/O) in the M/600. We first discuss the general operation of the host I/O subsystem, and then discuss I/O interrupts and the *Vector* instruction. Finally we extend our description of I/O to include the M/600 I/O processor, the Data Control Unit(s) and the M/600 basic I/O devices.

Host I/O

The M/600 can communicate with I/O devices using three methods: programmed I/O, data channel I/O, and high-speed channel I/O. Programmed I/O is used to transfer data to and from all slow I/O devices, such as terminals, and to control data channel devices. Only certain fast devices, such as discs and tape units, use data channel I/O, and only very fast devices use the burst multiplexor.

Device Codes

The M/600 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 respond only 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.

Busy and Done Flags

I/O devices are controlled by manipulating their Busy and Done flags (but note that data channel devices require several I/O instructions to set them up before they can be started with the flags). You can change the value of these flags using optional mnemonics appended to the instruction, corresponding to a code in bits 8 and 9 of the I/O instruction format. 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 one byte or 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. See *Programmer's Reference Manual - Peripherals (DGC No. 015-000021)* for details about programming specific devices in the I/O system.

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.

Data channel devices are controlled in two phases. Phase I specifies the parameters of the transfer, i.e., the starting location in memory and the number of words to be transferred. This is done with programmed I/O instructions. Phase II 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.

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.

For more information on the data channel, see *Programmer's Reference Manual - Peripherals (DGC No. 015-000021)* and *User's Manual - Interface Designer's Reference (DGC No. 015-000031).*

Burst Multiplexor I/O

The burst multiplexor transfers data directly between high speed devices and memory. The device controller performing the data transfer controls the burst multiplexor and its MAP. Except for setting up the burst multiplexor's map table, no program control is required and no CPU interaction is required. As a result, burst multiplexor data transfers are limited only by the memory speed. If the burst multiplexor and the CPU attempt to access memory at the same time, the burst multiplexor has priority.

The maximum data rate for the burst multiplexor is:

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

Address Translation Via the Burst Multiplexor

The burst multiplexor has two address modes; physical (unmapped) and logical (mapped). Since the burst multiplexor uses a different memory port from the CPU, it contains its own MAP. The differences between the two address modes, as well as the programming procedures for the MAP, are described in this section.

In the unmapped mode, the burst multiplexor receives 21-bit addresses from the device controllers, and passes them directly to memory. As each data word is transferred to or from memory, the address is incremented, causing successive words to move to/from consecutive locations in memory.

The mapped mode may be specified by a controller when it initiates a data transfer. In this case, the high order address bit is ignored, and the remaining 20 bits are used by the burst multiplexor as a logical address. The high order 10 bits of the logical address form a logical page number, which is translated by the MAP into an 11-bit physical page number. This page number, combined with the 10 low order bits from the logical address, forms a 21-bit physical address which is passed to memory.

The MAP translates logical page numbers to physical ones by using them as indices into its map table. The controlling program will have loaded this table with the desired values before I/O transfers begin. The table contains 1024 map registers, each of which holds an 11-bit physical page number. The logical page number is used 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 burst multiplexor performs a mapped transfer, and its address increments after each data word is moved, 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.

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 M/600 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.

I/O Interrupts

The I/O interrupt system in the M/600 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 interruptable 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, permitting 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 are controlled by 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.

Non-base 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.

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 interruptable 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 themselves are dynamically adjustable; you can change them as needed to fit a situation.

Processing an Interrupt

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. If 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* to location 0 returns control to the proper location in the interrupted program.

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,
- enable the user map if one was enabled before the interrupt,
- 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, to permit an easy return to processing the first interrupt return information 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.

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 - 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 has 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.

Note that using the faster modes gives you faster reaction time at the interrupt, but requires more careful design of the interrupt handler that eventually receives control. The interrupt handler must do what the *Vector* instruction did not take the time to do, i.e., store the contents of the carry bit and any accumulators it uses when servicing the interrupt. There are no problems doing this when using mode A, but mode B uses AC0, destroying the previous contents.

I/O Processor

The I/O Processor (IOP) is an ECLIPSE CPU that resides in the mainframe of a M/600 which is its *host*. It is controlled by the host and can take over jobs such as terminal scanning and servicing slow I/O devices. The IOP has its own I/O bus which is independent of the host's.

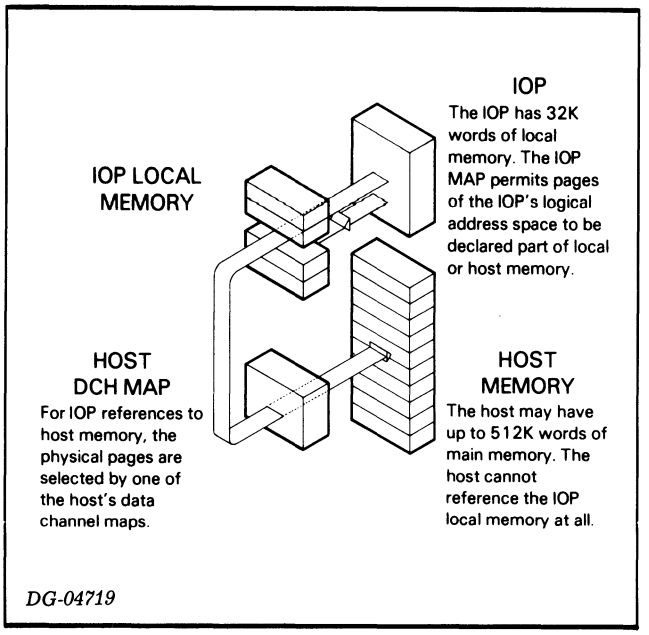
The IOP has 64K bytes of local memory with parity checking and testing available under software control. It executes the standard ECLIPSE instruction set, plus character manipulation instructions **CMV, CMP, CTR, CMT, ELDB,** and **ESTB,** and several special I/O instructions.

The processors can communicate with each other by programmed I/O. The host can perform all functions on the IOP which an operator can perform using the front panel of a normal CPU (except uINST).

The IOP has a MAP feature which enables pages of its address space to be mapped into host memory. Either processor can interrupt the other; the IOP also has a line-frequency timer which can be enabled to generate interrupts. Additionally, the IOP can produce a special micro-interrupt of the host to modify the host MAP.

Memory Addressing

The IOP has 64K bytes of local memory. This memory is not expandable, and its address range is fixed from 0 to 77777_8. It can be referenced only by the IOP and its data channel, not by the host. The IOP MAP feature divides its address space into 32 1K-word pages. Separate maps are maintained for the IOP and its data channel.

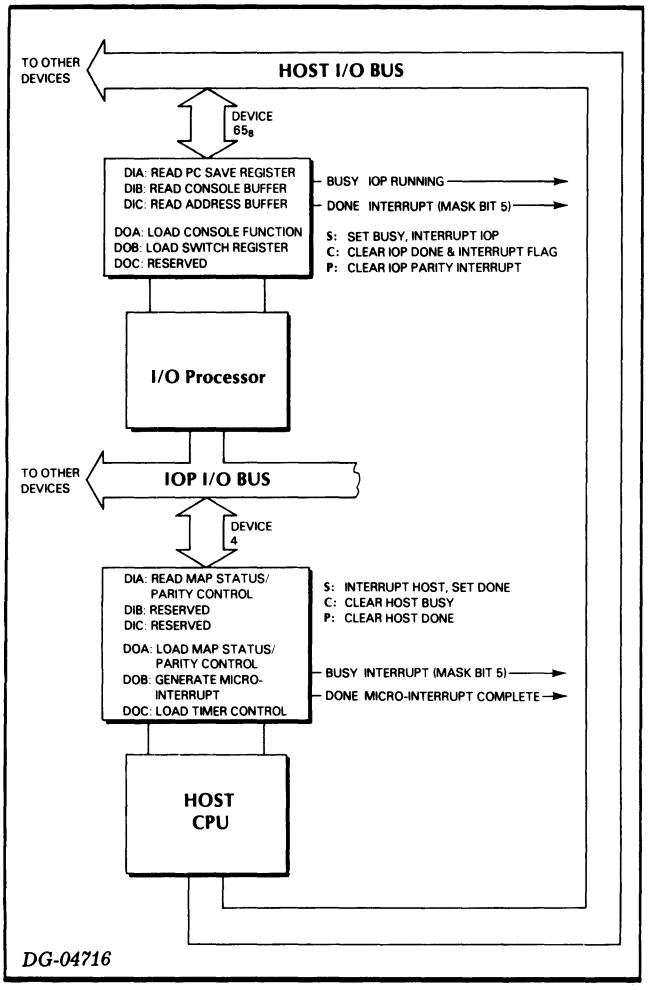


Bits in the IOP Map allow references to any page to be mapped into host memory through one of the host's four data channel maps. For IOP program references, one host map is selected by two bits in the map status register, which is accessible by software. For IOP data channel references, the two bits are provided by the device making the reference.

Setting the Parity Enable bit in the map status/parity control register will enable parity checking for the IOP local memory. Any occurence of a parity error will cause an IOP system reset and will interrupt the host. A parity Test feature is available which causes the IOP to produce incorrect parity on all memory writes. Any words so written will produce parity errors when read.

Host/IOP Communicaton

Each processor is accessible as an I/O device to the other. The IOP device code on the host I/O bus is jumper selectable to any value from 60_8 through 67_8, with the standard for a single-IOP system being 65_8. The host has device code 4 to the IOP. A cross interrupt facility allows each processor to interrupt the other. Either processor can disable interrupts from the other by setting bit 5 of its interrupt mask register with the **MSKO** instruction. The host's Busy flag from device 65_8 will be 1 whenever the IOP is running. The accompanying diagram illustrates the communication paths between IOP and host.



IOP Registers

The IOP has five registers which are accessible to the host. These registers allow the host to perform console functions on the IOP, as well as monitor its status and analyze error conditions. These registers include a switch register, console buffer register, console function register, PC save register, and an address buffer.

In addition, the IOP has a map status/parity control register which controls the parity and map features in the IOP map. It is also used to read back the host/local bits in the IOP map. This register is referenced from the IOP through device code 4. It is not accessible to the host.

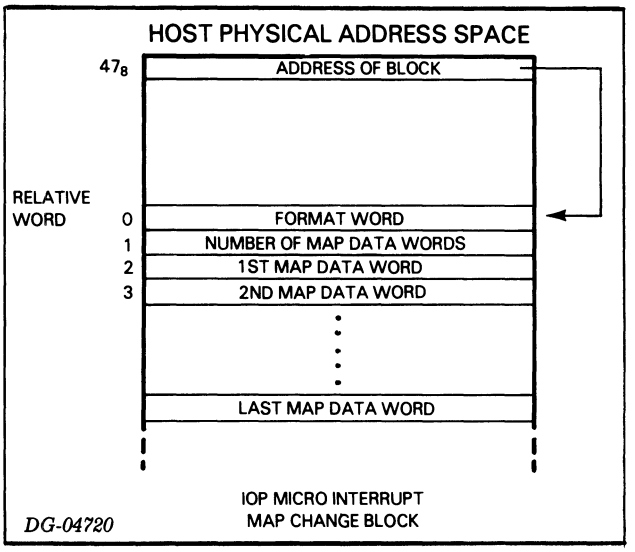
IOP Modifications to Host Map

The IOP can modify the host map tables without disrupting host operation; it does so using a block of data maintained in host memory. This feature, called the *micro interrupt*, means that the host program need not maintain a separate data channel map or maps to service the IOP; the host merely maintains data in memory that the IOP uses to generate the new map conditions. The IOP performs the micro interrupt by executing the *IOP generate micro interrupt* instruction which loads the MAP.

When the instruction is executed, physical location 47_8 in host memory must contain a pointer to a block of map data. The first word in the block is a format word which determines the particular map to be loaded. The second word in the block is a count of the number of map data words to be loaded, which must be no greater than 32. The words following the count contain the map data words.

When the IOP executes the *IOP generate micro interrupt* instruction, host program execution is suspended while the map data is loaded. The micro-interrupt is transparent to the host; there is no provision by which the host may mask out micro-interrupts, although they are disabled if the host turns off all interrupts. The host's Done flag (device 4 to the IOP) is set to 1 to signal the completion of a micro-interrupt.

The diagram that follows shows the format of the map data block.



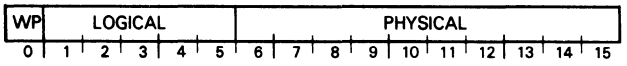
Map Format Word

As summarized by the table below, bits 6-8 of the format word determine the particular map to be loaded. All other bits of the format word are ignored.

Bits	Which map to load
000	User A
001	Reserved for future use
010	User B
011	Reserved for future use
100	Data channel A
101	Data channel B
110	Data channel C
111	Data channel D

Map Data Word:

The table below shows the format of the map data words.



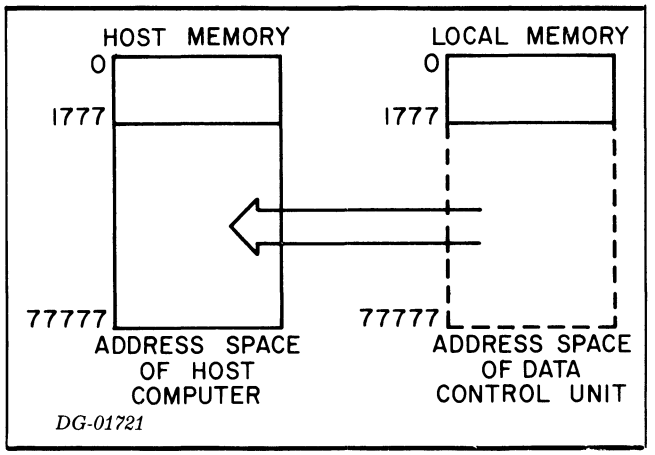
BITS	NAME	CONTENTS or FUNCTION
0	WP	Unused for data channel maps; write protect for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

Data Control Unit

The M/600 computer may have a number of data control units (DCU/50s) configured as part of its communications system. The DCU/50 is a user programmable controller with the power and flexibility of a general purpose processor. This section outlines the features of the DCU/50. For more detailed information on its operation, see Programmer's Reference, Data Control Unit *(DGC no. 015-00060)*.The DCU/50 executes a NOVA compatible instruction set and has independent input/output (I/O) capabilities. This allows preprocessing of I/O device functions for its main or *host* computer which may be either the M/600 Host computer or the IOP. It appears as an I/O device to its host, but shares much of the same memory via a data channel memory link.

Memory

A unique feature of the DCU/50 is its memory structure. The memory address space of the DCU processor is 32,678 16-bit words, including 1024 words of fast bipolar RAM memory local to the DCU processor and corresponding to the lower 1024 addresses of the DCU addressable memory. The remainder of the 32,678 words of DCU addressable memory are host processor memory. When the data control unit references an address within the host memory, the DCU processor is temporarily stopped while data is transferred into or out of the shared memory via the data channel of the host computer.



Programmed I/O Capabilities

The DCU/50 has its own I/O bus, which is both hardware and program compatible with the NOVA line. Like the NOVA I/O bus, the data control unit can address up to 64 unique device codes (2 of these are reserved), and includes program interrupt control and interrupt priority mask features. Unlike the NOVA, the DCU/50 supports only device controllers driven by program interrupts; the DCU/50 has no data channel.

Local I/O Devices

The DCU has a real-time clock and a data channel map select register as local I/O devices. It can also support any programmed I/O controllers compatible with the NOVA line. The devices on these controllers are not accessible to the host processor.

Host Processor Interface

The DCU/50 is an I/O device to the host computer. The host computer can reset, start, stop, and single-step the DCU processor. It can also place the DCU in diagnostic mode and request diagnostic data. In addition, the host and the data control unit can communicate via a cross interrupt facility. Data can also be passed from the host to the data control unit through a single Host To Data Control Unit (HTDCU) register.

BASIC I/O DEVICES

There are three I/O devices which are common to all M/600 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 M/600 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.

CONSOLE

The M/600 console is a powerful tool for creating and debugging programs. The state of the CPU, the floating point processor, and the I/O 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 burst multiplexor, 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 satellite 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 at the end of Chapter VII for complete documentation of the console facilities.

Using the Console Address Mode Feature

The M/600 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 M/600 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 32K of memory. (MD mode will cause the check to be performed in the 32K 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 doesn't detect an error, 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, 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 the negative 2's complement 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 M/600 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 hit 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/μInst switch down.
Push and hold the PLoad/Exec switch down.
Continue to push Inst/μInst 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.

COMMUNICATIONS SUBSYSTEMS

Data communications subsystems provide a reliable yet economic path for transferring digital information between a computer and some other device (which can be another computer). Usually, more than one other device and path is involved, along with a system of priorities for use when more than one device wants to communicate with the computer at once. Communications systems range from very small systems with just a few local lines to very large systems with lines extending many thousands of miles.

The M/600 computer supports a wide variety of communications devices in a number of configurations. This section introduces you to these configurations and the devices which can be supported.

Hardware Supported Configurations

There are two major types of communication configurations in the M/600 - host communications, and IOP communications.

Host Communications

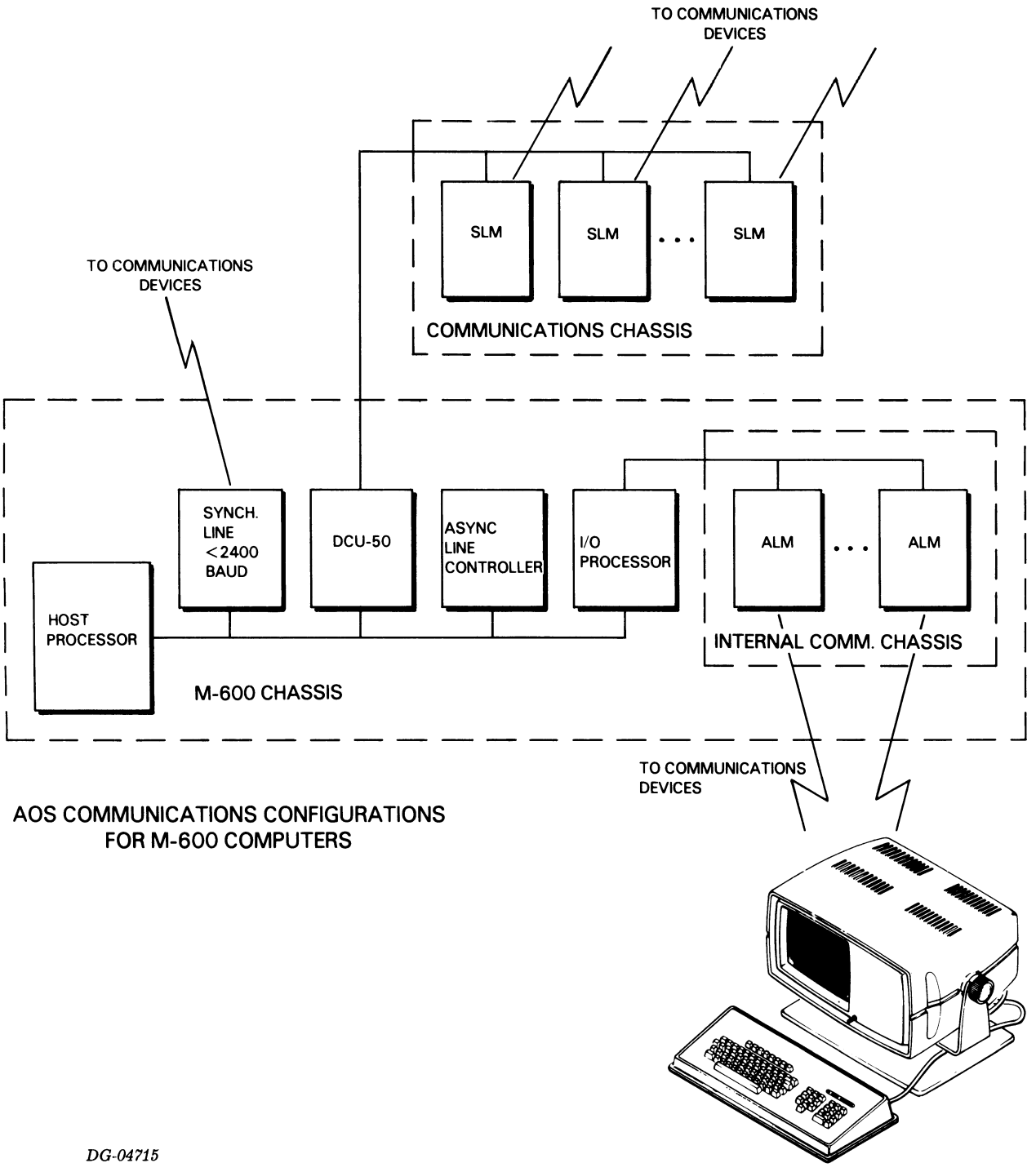
The Host computer can be configured with any Data General communications subsystem which has its roots in the host chassis and resides on the Host I/O bus. In addition to the basic I/O controller which is standard in the M/600 system, these subsystems include:

- 4010 series basic I/O controller as secondary device code
- 4060 series quad multiplexors
- 4074 synchronous line controllers
- 4241 series universal line multiplexors
- 4250 DCU/50-based DG/CS

I/O Processor Communications

The IOP can be configured with any Data General communications equipment including those devices which require a separate communications chassis. The unique feature of the IOP is that its I/O bus is connected directly to an 8-slot communications chassis which supports all DG/CS line multiplexors. The IOP supports communications systems in the M/600 main chassis which include:

- 4010 series basic I/O controllers
- 4060 series multiplexors
- 4074 synchronous line multiplexors
- 4241 series universal line multiplexors
- 4250 DCU/50-based DG/CS communications systems with external communications chassis.



AOS COMMUNICATIONS CONFIGURATIONS FOR M-600 COMPUTERS

DG-04715

In the M/600 local communications chassis, the IOP will support DG/CS equipment including:

- 4255 series asynchronous line multiplexors
- 4261 EIA (RS232-C/CCITT V-24)interface modules
- 4260 20ma current-loop interface modules
- 4263,4264 synchronous line multiplexors
- 4265 type 303 synchronous modem interfaces
- 4266 CRC generator

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

Software Support

The M/600 computer is supported by Data General's Advanced Operating System (AOS). Under this operating system, communications devices may be configured in a wide variety of ways. The list below and the figure which follows are intended as a brief outline of the systems available. Consult your Data General salesperson for more detailed information.

Host Communications

- 4010 series basic I/O controllers (as secondary device code)
- 4242 synchronous lines on the host I/O bus at an aggregate data rate up to 2400 baud
- 4250 DCU/50-based DG/CS configurations using synchronous lines only

I/O Processor Communications

- 4255 series asynchronous line multiplexors in the IOP communications chassis for a total of 8 boards (max).

ERROR CHECKING AND CORRECTION

The Error Checking and Correction (ERCC) facility is designed for applications where either a high degree of reliability is required for the main memory of a system, or where a graceful "fail-soft" capability is desired 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.

Method of Operation

The word length of an ERCC memory is 21 bits. These 21 bits are broken into 16 data bits followed by 5 ERCC bits. This check field is constructed by a hardware encoder from the 16 data bits and is written each time the memory location is written into. When the memory location is read, the encoder checks the ERCC bits read from memory. If the 21 bits do not generate an error code, the 16 data bits are passed on to the CPU. Otherwise, a single bit error has occurred. The memory pauses while the single bit in error is corrected and the entire corrected word is rewritten into the memory location. The data is then passed on to the CPU and the ERCC facility 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.

The logic of the ERCC facility is such that all single-bit errors are detected and corrected. In the rare event that a multi-bit error occurs, either it is detected and reported as such with no correction, or it is incorrectly interpreted as a single-bit error and that bit is complemented.

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 the IOP and any DCU/50s in the system are semiconductor so the restart facility must reload those memories before restarting the processors.

This page intentionally left blank.

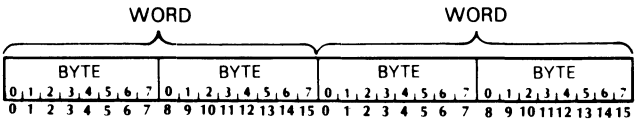
Chapter III
DATA AND INSTRUCTION FORMATTING

DATA FORMATS

In this chapter, we describe the formatting conventions we use for entering data and instructions in the M/600. Some of the conventions used require a knowledge of the binary, octal, and hexadecimal numbering systems. See Appendix D for a review of these numbering systems.

Bit Numbering Convention

In representing bits in a byte or in a computer word, we number them from left to right, with the leftmost (high-order) bit always numbered 0. We always number bits using the decimal numbering system.

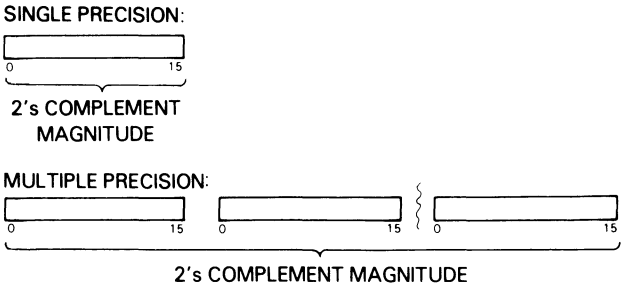


Integer Format

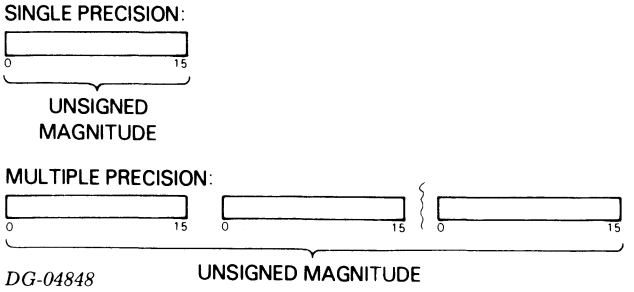
We represent a signed integer by a two's-complement number in one or more 16-bit words. For a single precision signed integer in two's complement notation the range of values that you can represent is 100000_8, or -32,768 (the smallest value) through 077777_8, or 32,767 (the largest value). The value of a two's complement signed integer is always greater than or equal to zero if its high order bit is 0; the integer is always less than zero if its high order bit is 1.

We represent an unsigned integer by a binary quantity using all the bits of one or more 16-bit words. For a single precision unsigned integer, the range of values that you can represent is 000000_8, or 0 (the smallest value) through 177777_8, or 65,535 (the largest value). Unsigned integers are always greater than or equal to zero.

SIGNED INTEGERS



UNSIGNED INTEGERS



DG-04848

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 *Carry* bit. A change in the value of the carry bit indicates a carry out during fixed point arithmetic operations.

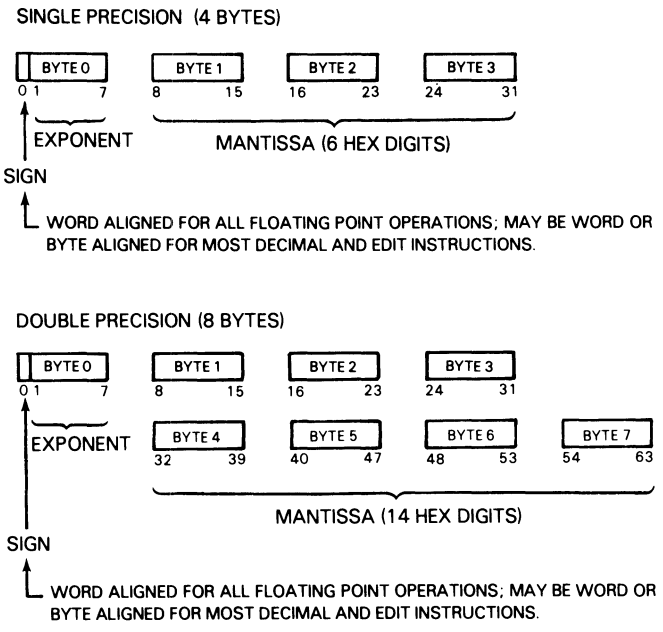
Floating Point Format

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 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*, that is, 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.



DG-04849

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.

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.

Byte Format

We represent bytes as 8-bit unsigned binary integers. A byte in memory is selected by a 16-bit *byte pointer*. (See *Addressing*, below.) Byte format is used when manipulating bytes, particularly alphanumeric characters.

Decimal Format

We represent decimal numbers by a variety of industry-compatible formats. Both *unpacked* and *packed* decimal format can be recognized and manipulated by various instructions.

Unpacked Decimals

In unpacked decimal format, each byte of memory contains the code for one ASCII character. Each decimal digit is represented by the ASCI character for that digit except when a digit and sign are combined in one character. The table below shows the ASCII characters we use to represent the combination of a digit and sign in those formats which require it.

Digit	Digit With + Sign		Digit With - Sign	
	ASCII Character	Octal Code	ASCII Character	Octal Code
0	{	173	}	175
1	A	101	J	112
2	B	102	K	113
3	C	103	L	114
4	D	104	M	115
5	E	105	N	116
6	F	106	O	117
7	G	107	P	120
8	H	110	Q	121
9	I	111	R	122

You can represent the sign in any one of four ways when using unpacked decimal format. These four ways are shown in the table that follows.

Note that in each example, the first line shows the decimal number as normally written, the second line shows the ASCII characters placed in each byte, and the third line shows the octal code of the character in each byte.

Type	Characteristic	Example
Leading Sign	Sign appears in separate byte before number.	+2048 + 2 0 4 8 053 062 060 064 070
Trailing Sign	Sign appears in separate byte after number.	-1756 1 7 5 6 - 061 067 065 066 055
High-order Sign	Sign and high-order digit are indicated by single (first) byte.	+1850 A 8 5 0 101 070 065 060
Low-order Sign	Sign and low-order digit are indicated by single (last) byte.	-3972 3 9 7 K 063 071 067 113

Packed Decimal

In packed decimal format, each digit of the decimal number occupies one half byte in memory. The sign appears in a separate trailing half byte. The number must start and end on a byte boundary, so a packed decimal number always consists of an odd number of digits followed by the sign (a zero is placed in front of numbers with an even number of digits). The sign is represented by the octal number 14_8 for plus and 15_8 for minus.

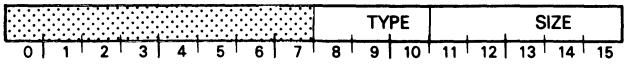
Several examples of packed decimal numbers are shown below.

	BYTE		BYTE		BYTE	
+2048	0 00	2 02	0 00	4 04	8 10	+ 14
+32,456	3 03	2 02	4 04	5 05	6 06	+ 14
-1756	0 00	1 01	7 07	5 05	6 06	- 15
-25,989	2 02	5 05	9 11	8 10	9 11	- 15

Data Type Indicator

Most M/600 instructions make certain assumptions about the representation of data in memory -- whether the data you are referencing is in integer format, floating point format and so on. The assumptions about data type made by the instructions are usually obvious; your choice of instruction implicitly defines the kind of data you are manipulating. For example the *Load byte* assumes the information to which you refer is a single byte of data, while the *Load floating point double* instruction operates on an aggregate of data in memory that is eight bytes long.

However, the decimal arithmetic and the edit instructions do not make such assumptions; rather, these instructions require you pass them a parameter called the *data-type indicator* which defines both the data representation you want the operation to use and also its size; you pass the indicator in an accumulator. The data-type indicator has the following format:



BITS	NAME	CONTENTS or FUNCTION
0-7	---	Reserved for future use
8-10	TYPE	Data type: 0 Unpacked decimal, low order sign 1 Unpacked decimal, high order sign 2 Unpacked decimal, trailing sign 3 Unpacked decimal, leading sign 4 Unpacked decimal, unsigned 5 Packed decimal 6 Two's complement integer, byte aligned 7 Floating point, byte aligned
11-15	SIZE	Data length: For all except data type 5, count of bytes in number *minus 1* (including sign); For data type 5, the count of *digits* in the number

ADDRESSING CONVENTIONS

The various methods of addressing memory locations in the M/600 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 an address space of 15 bits. This is a reference to the *logical* address space - the address space which the user normally sees and 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.

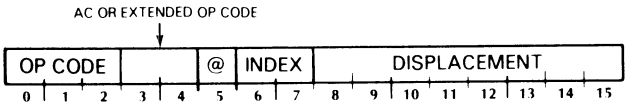
The MAP controls the relationship between a logical address space and the physical address space by translating logical addresses to physical addresses. It has no effect on addressing from the programmer's point of view, unless the MAP itself is being programmed.

When the MAP is enabled, it intercepts memory references and translates the 15-bit logical address into a 20-bit physical address. The translation functions are programmed into the MAP, but the translation process is invisible to the regular user.

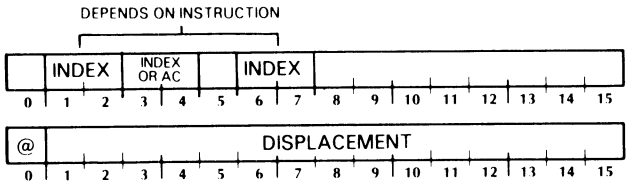
Word Addressing Definitions

The following definitions are useful for understanding word addressing in the M/600 :

SHORT CLASS:



EXTENDED CLASS:



Addressing Modes - Three methods of addressing using a displacement from some reference point to find the desired address. Different modes use 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. Below 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 10 11	-200_8 to 177_8 or -128 to $+127_{10}$	-40000_8 to 37777_8 or $-16{,}384$ to $+16{,}383_{10}$

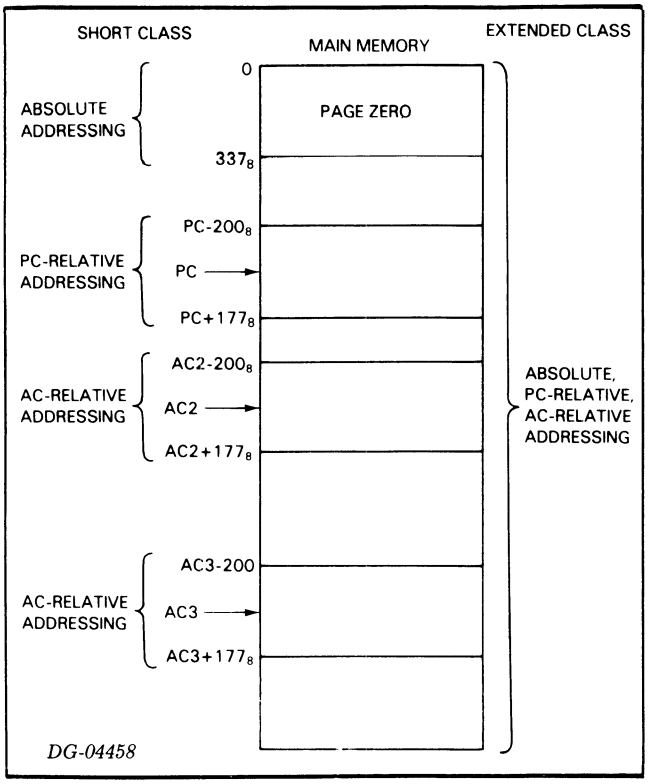
Addressing Modes

Word addressing in the M/600 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 effective 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 any word in 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 these locations for special purposes.

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

P.C. Relative Addressing Mode - In P.C. relative addressing mode, the effective 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 effective address is found by adding the displacement to the contents 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 also 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 M/600 , but indirect protection is available which can limit indirections to 15 levels (see the MAP section, Chapter III).

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.

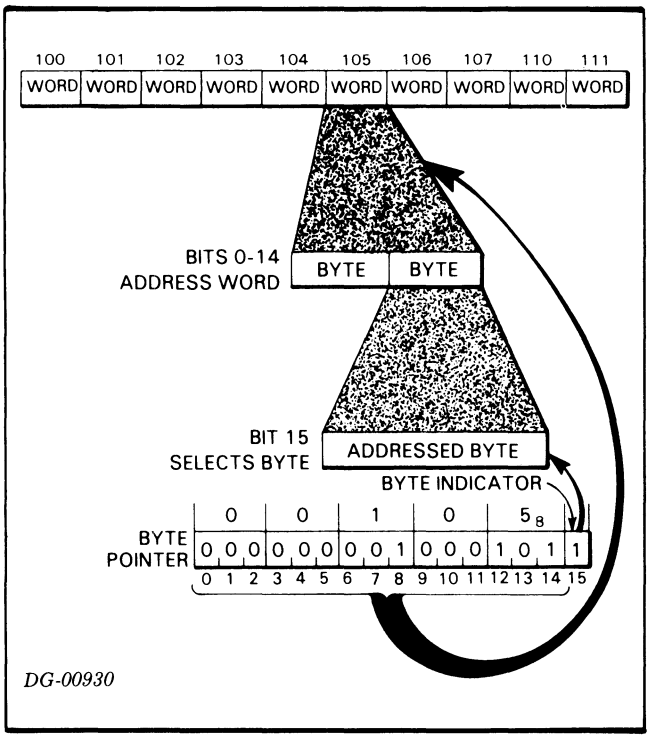
If the intermediate address of a short class instruction is in the range $30\text{-}37_8$, and the indirect bit is 1, the contents of the addressed location are decremented by one, and the addressing chain continues using the *decremented* value of the addressed location. Within CPU Map A and Map B logical address spaces, and when demand paging is enabled, indirection chains pass through locations 20_8 -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 address in the chain, rather than the effective address.*

You can find a flow diagram of the addressing process in an appendix.

Byte Addressing

We use a 16-bit *byte pointer* to address bytes in memory. 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.



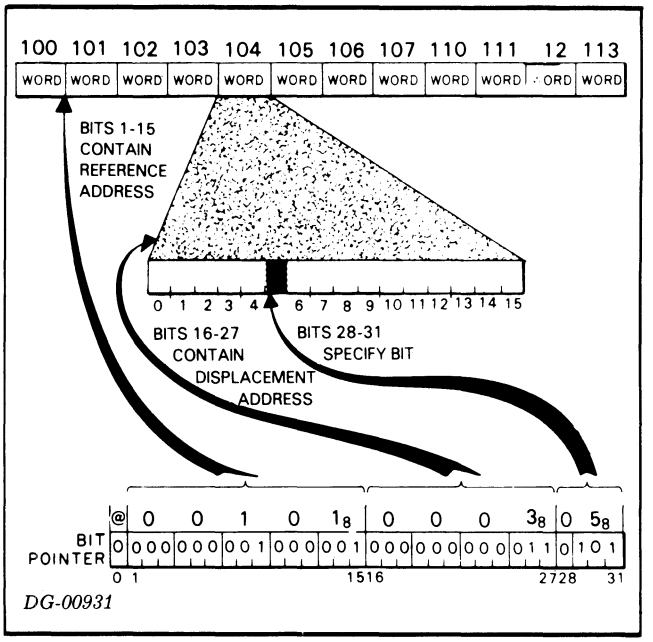
Bit Addressing

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

Below is a diagram of the bit-addressing process.



RESERVED STORAGE LOCATIONS

There are 36 reserved storage locations in the M/600. 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)
47	MAP CHANGE DATA BLOCK POINTER	Address of the block of map data used by the IOP during micro interrupt (not indirectable)

The following locations may be in unmapped logical address space or in Map A or Map B logical address space. They are usually placed in unmapped logical address space:

Loc	Name	Function
4	VECTOR STACK POINTER	Address of the top 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 four locations are in Map B logical address space:

Loc	Name	Function
10	PAGE FAULT CONTEXT BLOCK ADDRESS	Address in MAP B of the top of the page fault context block (indirectable)
11	PAGE FAULT HANDLER ADDRESS	Address in MAP B of the page fault handler (indirectable)
12	BREAKPOINT HANDLER ADDRESS	Address in MAP B of the breakpoint handler (indirectable)
13	VALIDITY BIT POINTER	Address in MAP B of the validity bit pointer (indirectable)

The following locations are usually in Map A or Map B logical address space, but they may be in unmapped addressed space too:

Loc	Name	Function
20-27[1]	AUTO-INC0 through AUTO-INC7	Auto-incrementing locations[1]
30-37[1]	AUTO-DEC0 through AUTO-DEC7	Auto-decrementing locations[1]
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	DECIMAL/EDIT FAULT ADDRESS	Address of the decimal/EDIT fault handler (indirectable)

[1] *Auto-incrementing and auto-decrementing is suppressed in CPU Map A and Map B logical address spaces whenever demand paging is enabled.*

PROGRAM EXECUTION

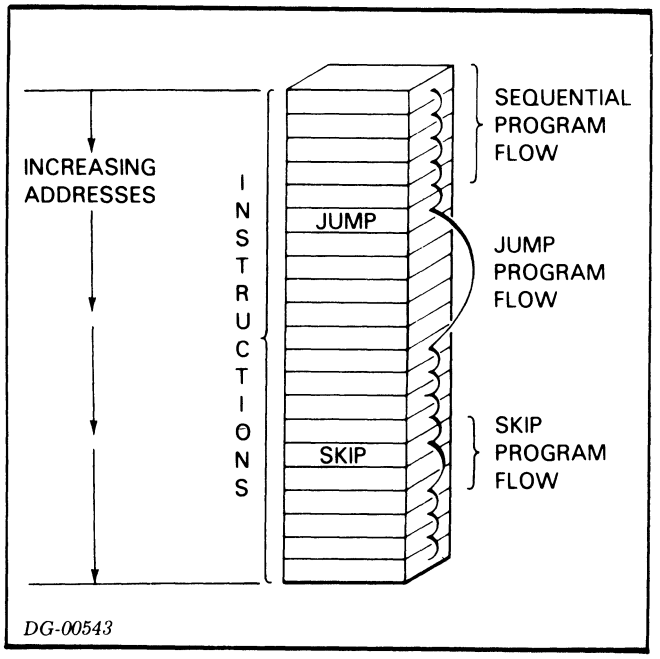
Sequential Operation

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

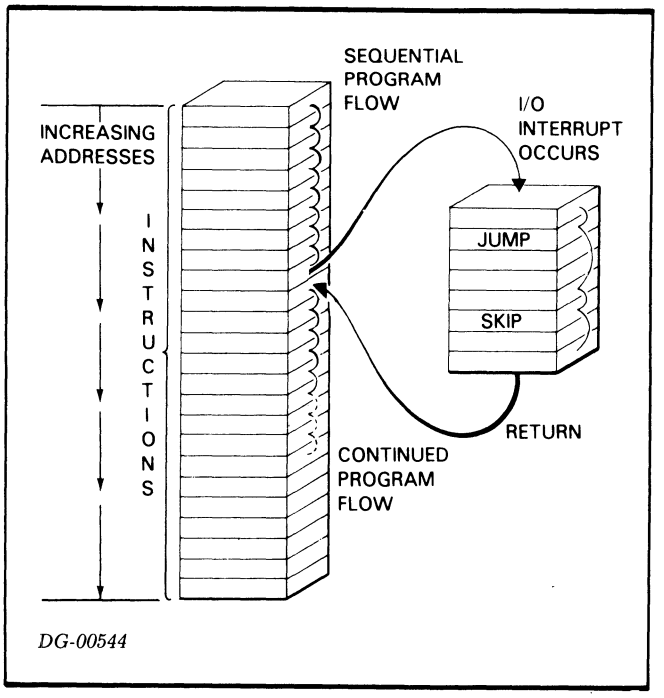
You can alter the program flow from sequential operation in two ways. Jump instructions alter the program flow by inserting a new value into the program counter. Conditional skip instructions alter the program flow by incrementing the program counter an extra time if a specified test condition is true. 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 will cause a 1-word skip which will result 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 address of the next sequential instruction in the interrupted program 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.



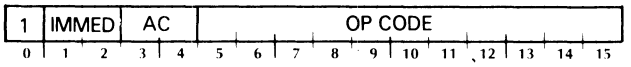
INSTRUCTION FORMAT

The format for each instruction is shown at the instruction description. Most of the formats are quite simple and need no additional explanation. Those that do are discussed in this section.

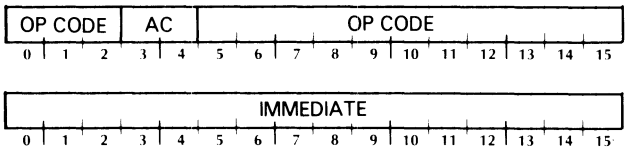
Below are diagrams of the formats which are discussed below.

IMMEDIATE FORMAT

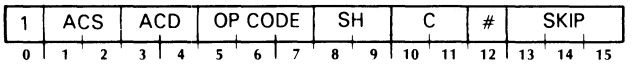
SHORT CLASS:



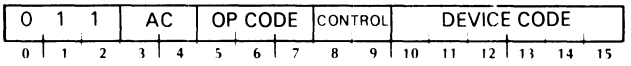
EXTENDED CLASS:



TWO ACCUMULATOR-MULTIPLE OPERATION FORMAT



I/O FORMAT



Memory Address Formats

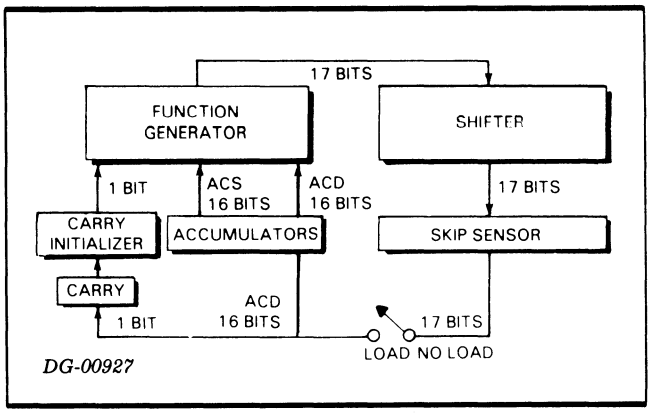
There are four types of memory address formats - two with and two without an accumulator reference. The memory address portion of the format is identical, however, and was discussed in detail in the section on Addressing Conventions. When applicable, include the identity of the accumulator as indicated in the format diagrams.

Immediate Formats

Sometimes it is more efficient to obtain a constant by using an *immediate* rather than retrieving it from memory. An immediate is a constant which is stored in the instruction word itself (short class of instructions), or in the next memory word (long class), and is coded along with the instruction.

ALC Format

The Arithmetic/Logical/Conditional (ALC) instructions use an arithmetic logic unit which can perform several operations on two data words, using one instruction. The logical organization of the arithmetic logic unit is illustrated below.



The function generator provides 8 functions: *Add, Subtract, Negate, Add Complement, Move, Increment, Complement,* and *AND.*

All the instructions using this arithmetic unit operate on the contents of 1 or 2 accumulators and the carry bit. These instructions specify a shift operation, which is performed by the shifter in the arithmetic unit. The shifter operates on the 17-bit quantity consisting of the 16-bit output of the ALU and the carry bit. The shifter can rotate this quantity left or right, or swap the two bytes in the accumulator.

The arithmetic logic unit can test the result of the function generator/shifter combination for various conditions, and skip or not skip the next instruction depending on the results of the test.

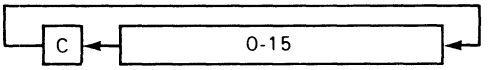
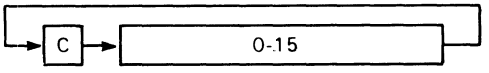
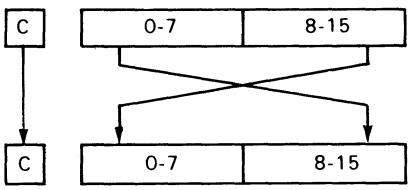
If the no load bit is 1, the results of the shift operation are not loaded into the destination accumulator, but all the other operations (such as skip tests) take place.

> **NOTE:** *These instructions must not have both the* No-Load *and the* Never-Skip *options specified at the same time. These bit combinations are used by other instructions in the instruction set.*

The following table summarizes the various operations that can be performed by all ALC instructions.

SYMBOL	VALUE	OPERATION
[c] omitted	00	Leave Carry bit unchanged
[c]=Z	01	Initialize Carry bit to 0
[c]=O	10	Initialize Carry bit to 1
[c]=C	11	Complement the Carry bit
[sh] omitted	00	Do not shift the result of the ALC operation
[sh]=L	01	Rotate left the 17-bit combination of Carry bit and ALC operation result
[sh]=R	10	Rotate right the 17-bit combination of Carry bit and ALC operation result
[sh]=S	11	Swap the two 8-bit halves of the ALC operation result without affecting Carry bit
# omitted	0	Load the result of the shift operation into ACD
#	1	Do not load the ALC operation result into ACD; restore Carry bit to value it had before shifting
[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

The following diagrams illustrate the operation of the shifter.

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

I/O Instruction Format

The I/O instructions control the transfer of data between the computer and the various I/O devices by manipulating the Busy and Done flags in each device (see the discussion of I/O facilities in Chapter II).

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

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

The next table applies to the I/O SKIP instruction which tests the Busy and Done flags.

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

The last table applies to I/O instructions with a device code of 77_8. These instructions are used by the Interrupt System and for special CPU functions. Instead of manipulating or testing the Busy and Done flags, these instructions operate on the Interrupt On and Power Fail flags.

SYMBOL	VALUE	OPERATION
[f] omitted	00	Does not alter the Interrupt On flag
[f]=S	01	Sets Interrupt On flag to 1
[f]=C	10	Clears Interrupt On flag to 0
[f]=P	11	Leaves Interrupt On flag unchanged (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

Chapter IV
M/600 INSTRUCTION SETS

In this chapter, we introduce the various instruction sets of the M/600. Each section describes the general characteristics of the instructions in that instruction set, along with any special properties of those instructions. A table at the end of each section summarizes the action of all the included instructions.

Because the catagories for instruction sets are somewhat arbitrary - their boundaries are often diffuse and open to argument - we include many instructions in more than one instruction set. Thus, the *Load Floating Point* instruction appears in both the Memory Reference instruction set and the Floating Point instruction set.

For the full description of any particular instruction, refer to its entry in Chapter V -- or, if it is an I/O instruction for a specific device (e.g., the MAP, the IOP/Host interface, etc.) refer to Chapter VI. We've placed the instructions in both chapters in alphabetical order, by instruction mnemonic in Chapter V and by I/O device mnemonic in Chapter VI.

MEMORY REFERENCE

44 M/600 memory reference instructions perform one of the following operations:

- Load data from memory to a machine register;
- Store data from a machine register to memory;
- Move data from one area of memory to another;
- Alter the contents of memory.

Typically, the instructions use a 15-bit memory address, a 16-bit byte pointer or a 32-bit bit pointer to reference memory. Alternatively, they can construct an *effective address* from a displacement contained in the instruction. Some instructions allow indirection; that is, the instruction traces a chain of pointers through memory before performing its data operation. (Information addressing is described in Chapter III.) The action of all these instructions is restricted to the logical address space of the computer.

Among those instruction which use the machine registers, some can reference a variety of registers. Those instructions use information coded in the instruction to select the intended register(s). Other memory reference instructions use one or more specific registers by convention. In all cases, the machine registers may be either sources or destinations of data or they may contain control information that directs the action of the instruction.

Memory reference instructions move or manipulate data in a variety of formats. Some instructions interpret the 16-bit contents of a memory address as a full word while others use a portion of the word - either an 8-bit byte (half word) or some individual bit within the word. Still other instructions combine one memory word with one or more other 16-bit quantities (often, at adjacent memory locations) and interpret the aggregate as a single-precision or double-precision floating point number or as a decimal number.

Nearly all of the memory reference instructions are shown in the following table. The *Edit* subprogram instructions are described separately. Note that there are several instructions which appear in both a short and a long form. The short form instructions are sixteen bits in length and can directly specify a memory address from 0-255 or they 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.

MEMORY REFERENCE INSTRUCTIONS

Mnem	Name	Function
BAM	Block Add And Move	Moves blocks of memory words from one location to another, adding a constant to each one.
BLM	Block Move	Moves blocks of memory words from one location to another.
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 translated string.
DSZ, EDSZ	Decrement And Skip If Zero	Decrements the addressed word, then skips if the decremented value is zero.
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.
FLDS, FLDD	Load Floating Point	Moves a word out of memory into a specified FPAC.
FLMD	Float From Memory	Converts the contents of two memory locations to floating point format and places the result in a specified FPAC.
FLST	Load Floating Point Status	Moves the contents of two specified memory locations to the FPSR.
FPOP	Pop Floating Point State	Pops an 18-word floating point return 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 return block onto the user stack.
FSST	Store Floating Point Status	Moves the contents of the FPSR to two specified memory locations.
FSTS, FSTD	Store Floating Point	Stores the contents of a specified FPAC into a memory location.
ISZ, EISZ	Increment And Skip If Zero	Increments the addressed word, then skips if the incremented value is zero.
LDA, ELDA	Load Accumulator	Moves a word out of memory and into an accumulator.
LDB, ELDB	Load Byte	Moves a byte from memory to an accumulator.
LDI LDIX	Load Integer	Converts a decimal integer in memory to floating point form and places it in an FPAC.

MEMORY REFERENCE (Continued)

Mnem	Name	Function
LEF, ELEF	Load Effective Address	Places an effective address in an accumulator.
LMP	Load Map	Loads successive words from memory into the MAP where they are used to define a user or data channel map.
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 onto the stack.
PSHJ	Push Jump	Pushes the address of the next sequential instruction onto the stack, computes the effective address *E* and places it in the program counter.
PSHR	Push Return Address	Pushes the address of the instruction after the next sequential instruction 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.
STA, ESTA	Store Accumulator	Stores the contents of an accumulator into a memory location.
STB, ESTB	Store Byte	Moves the right byte of one accumulator to a byte in memory.
STI, STIX	Store Integer	Converts the contents of an FPAC to a specified data type and stores it in a memory location.

LOGICAL OPERATIONS

There are 16 M/600 instructions which apply to logical operations. These instructions:

- Perform logical operations on operands in accumulators;
- Load a number into an accumulator.

Some logical operations instructions can address any of the four accumulators, using information coded with the instruction to select the intended accumulator(s). Those instructions that address two accumulators can use the same accumulator for source and destination.

Three of the logical operations instructions operate on 32-bit (2 accumulator) operands, and the rest operate on 16-bit operands. See Chapter III for more information about data formats.

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.

FIXED POINT ARITHMETIC

There are 26 M/600 instructions which perform fixed point arithmetic. These instructions:

- Perform binary arithmetic on operands in accumulators;
- Load data from memory to an accumulator;
- Move data from an accumulator to memory;
- Load a number into an accumulator.

Some fixed point arithmetic instructions can address any of the four accumulators, using information coded with the instruction to select the intended accumulator(s). Those instructions that address two accumulators can use the same accumulator for source and destination. Other fixed point arithmetic instructions use one or more accumulators by convention.

The fixed point instruction set operands are 16 or 32 bits in length and can be either signed or unsigned. See Chapter III for more information about data formats.

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 usually is indicated by the prefix *E* in the mnemonic). Most of these are instructions that move data to or from memory. For these, 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. **ADI** and **ADDI** are also short and long forms, respectively, of the same instruction. The short form can only add a 2-bit quantity coded with the instruction (an *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 an unsigned integer to the logical complement of another unsigned integer.
ADD	Add	Adds contents of one accumulator to another.
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 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.

BIT MANIPULATION

There are eight M/600 instructions which manipulate bits. These instructions:

- Locate a bit in memory and set it to 0 or 1;
- Test a bit, skipping the next word if the specified conditions are 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.

Bit instructions can address any two of the four accumulators, using information coded with the instruction to select the intended accumulators. The same accumulator can be used for both accumulators.

Five of the bit instructions use a bit pointer to locate a bit in memory. See Chapter III for a discussion of bit pointers. The other three bit instructions 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 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 was originally 0.

BYTE/CHARACTER MANIPULATION

There are nine M/600 instructions which process data in 8-bit bytes. These instructions:

- Load a byte from memory and place it in an accumulator;
- Store a byte from an accumulator to memory;
- Compare two strings of bytes in memory and return a comparison code;
- Translate a string of bytes from one data representation to another;
- Move a string of bytes from one area of memory to another;

All the instructions in this set use one or more 16-bit byte pointers to specify the memory address of an 8-bit datum. (See Chapter III for information about byte pointers.)

Instructions that move single bytes between an accumulator and memory (either direction) are of two types: some carry within the instruction a byte pointer to memory, and the others use an accumulator to hold the byte pointer. When an instruction moves a byte to an accumulator it also clears the high order half of the destination register. When an instruction moves a byte from an accumulator to memory, it leaves unchanged the other byte stored at the same memory address.

Among this set are instructions that perform functions on two groups of bytes or on two areas in memory. They compare the two strings on a byte by byte basis; they translate one string to another representation; they copy a string from one area to another; or they perform a combination of these functions. These instructions read control parameters from the accumulators defining source and destination byte pointers. Some read a specific string length from an accumulator at the start of an operation; others analyze the bytes being moved to determine when the instruction should terminate.

All of the byte instructions are shown in the following table. Some of them have both a long and short form, which differ in the effective range of byte pointer the instruction can generate. Most of the byte instructions are 16 bits long; but there are two of them that are 32 bits long.

Byte-Character Manipulation Instructions

Mnem	Name	Function
CMP	Character Compare	Compares one string of characters to another.
CMT	Character Move Until True	Moves bytes from one area of memory to another until the string runs out or reaches a delimiter.
CMV	Character Move	Moves a string of bytes from one area in memory to another.
CTR	Character Translate	Translates a string of bytes from one data representation to another.
EDIT	Edit	Converts a decimal integer to a string of bytes controlled by an edit subprogram.
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.

PROGRAM FLOW ALTERATION

There are 55 M/600 instructions that control program flow alteration - either unconditionally or on the basis of testing some logical condition in the machine. These instructions:

- Alter the normally sequential program flow by placing a new value in the program counter;
- Test a specific value and skip the next sequential word if the test result is true;
- Compare two values and skip the next sequential word if the test result is true;
- Compare a value with two other values and skip the next sequential word if the test result is true.

Some of these instructions can address any of the four accumulators, using information coded with the instruction to select the intended accumulator(s). Those instructions that address two accumulators can use the same accumulator for source and destination.

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. The last table summarizes the skip tests available for the ALC instructions (**ADD, ADC, AND, COM, INC, MOV, NEG** and **SUB**).

The third table summarizes skip instructions that test 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 skips if first 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.
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.
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 SVC	System Call	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.

I/O Skip Tests

SYMBOL	FUNCTION
[t]=BN	Tests Busy flag for nonzero
[t]=BZ	Tests Busy flag for zero
[t]=DN	Tests Done flag for nonzero
[t]=DZ	Tests Done flag for zero

ALC Skip tests

SYMBOL	FUNCTION
[skip] omitted	No skip
[skip]=SKP	Skip unconditionally
[skip]=SZC	Skip if Carry bit is zero
[skip]=SNC	Skip if Carry bit is nonzero
[skip]=SZR	Skip if ALC result is zero
[skip]=SNR	Skip if ALC result is nonzero
[skip]=SEZ	Skip if either ALC result or Carry bit is zero
[skip]=SBN	Skip if both ALC result and Carry bit is nonzero

FLOATING POINT ARITHMETIC

The floating point instruction set performs rapid arithmetic operations on numbers with a much larger range than the fixed point instruction set can feasibly handle. Single-precision floating point operations can handle about 7 significant decimal digits, while double-precision operations can handle about 16 significant decimal digits.

NOTE: *The M/600 floating point instructions assume normalized input numbers. Results are undefined for unnormalized input.*

Floating Point Registers

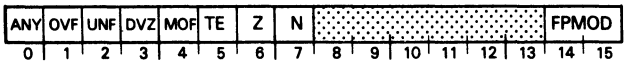
There are five registers available to the programmer in the floating point processor. These are the four floating point accumulators (FPAC's) and the Floating Point Status Register (FPSR). The FPAC's are numbered 0-3 and are called FAC0, FAC1, FAC2, and FAC3. The FPSR is a 32-bit register that contains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

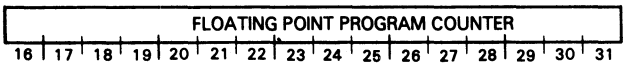
Guard Digit

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.





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.
8-13	---	Reserved for future use.
14-15	FPMOD	Indicates computer series supporting the floating point instruction set. 00 S/200, C/300, S/230, C/330 01 S/130 Series 10 M/600 Series 11 Reserved for 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.

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. The fault 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	Moves 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	Moves 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.
FMOV	Move Floating Point	Moves the contents of one FPAC to another FPAC.

FLOATING POINT (Continued)

Mnem	Name	Function
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	The next sequential word is executed.
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	The next sequential instruction is skipped.
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 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.
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	Moves the contents of the FPSR to two memory locations.
FSTS, FSTD	Store Floating Point	Stores 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.

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.

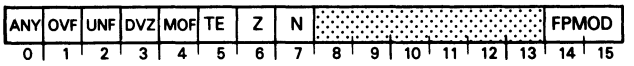
All the floating point functions are interruptable and interrupted instructions are restarted after the interrupt. As a result, certain Real Time Clock or Programmable Interval Timer frequencies may cause looping when you are evaluating very large polynomials with the *Polynomial evaluation* function.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

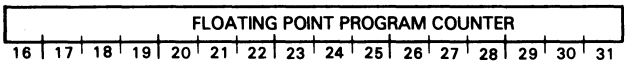
All the floating point functions are shown in the following table. Some instructions have two forms. The form using a mnemonic ending in **S** produces a single-precision floating point result while the form using a mnemonic ending in **D** produces a double-precision floating point result.

FLOATING POINT FUNCTIONS

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 undefined.
2	UNF	Underflow Indicator - while processing a floating point number, an exponent underflow occurred; the result is undefined.
3	DVZ	Divide by Zero - while processing a floating point number, a zero divisor was detected; division was aborted and the result is undefined.
4	MOF	Mantissa Overflow - not used by floating point functions.
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.
8-13	---	Reserved for future use.
14-15	FPMOD	Indicates computer series supporting the floating point instruction set. 00 S/200, C/300, S/230, C/330 01 S/130 Series 10 M/600 Series 11 Reserved for 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.

DECIMAL ARITHMETIC

There are 11 instructions in the M/600 which perform operations on decimal data. These instructions:

- Add and subtract decimal integers;
- Shift the contents of words one or more hex digits left or right;
- Convert decimal integers to floating point numbers;
- Convert floating point numbers to decimal integers of a specified data type;
- Convert decimal integers to strings of bytes and perform a variety of functions on the string.

Decimal integers occupy either a byte or half a byte, depending on the data type being used. See Chapter III for a complete discussion of the decimal integer data types used in the M/600.

The *Decimal add* and *Decimal subtract* instructions each use two of the four 16-bit accumulators, using information coded with the instruction to select the intended accumulators. The *Edit* instruction uses all four accumulators for dedicated purposes. The other decimal arithmetic instructions use a combination of 16-bit accumulators and 64-bit floating point accumulators.

Decimal Faults

In the course of processing decimal instructions, the CPU performs certain checks on the data being processed. If an invalid data type or number is found, a fault is initiated. When a fault occurs, the processor first pushes a return block onto the stack with the program counter word in the return block pointing to the instruction that caused the fault. It then places a code indicating the type of fault in AC1, and executes a *Jump indirect* to the decimal fault address, location 46_8. This location should point to a fault handling routine.

The table below describes the decimal faults:

CODE	INSTR.	MEANING
4	LDI STI STIX	Number too large to convert to specified data type. SI/DI is in AC2.
6	LSN LDI LDIX	Sign code is invalid for this data type. AC3 contains SI.
7	LSN LDI LDIX	Invalid digit. AC2 contains SI.

DECIMAL ARITHMETIC INSTRUCTIONS

Mnem	Name	Function
DAD	Decimal Add	Adds together the decimal digits found in bits 12-15 of two accumulators.
DHXL	Double Hex Shift Left	Shifts the 32-bit contents of two accumulators left 1 to 4 hex digits.
DHXR	Double Hex Shift Right	Shifts the 32-bit contents of two accumulators right 1 to 4 hex digits.
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.
EDIT	Edit	Converts a decimal integer to a string of bytes controlled by an edit subprogram; or manipulates string of bytes.
HXL	Hex Shift Left	Shifts the contents of an accumulator left a number of hex digits.
HXR	Hex Shift Right	Shifts the contents of an accumulator right a number of hex digits.
LDI	Load Integer	Converts a decimal integer to normalized floating point form and places it in a specified floating point accumulator.
LDIX	Extended Load Integer	Distributes a decimal integer into 4 floating point accumulators.
LSN	Load	Evaluates a number in memory and returns a code indicating the sign of the number.
STI	Store Integer	Converts the contents of a floating point accumulator to a specified format and stores it in memory.
STIX	Extended Store Integer	Converts the contents of 4 floating point accumulators to integer form and uses the 8 low-order digits of each to form a 32-bit integer.

ALPHABETIC AND NUMERIC FIELD EDITING

There are 24 instructions which perform alphabetic and numeric field editing. These instructions:

- Move characters from one string to another;
- Insert characters into a string;
- Convert data formats.

These instructions use all four accumulators.

This instruction set uses a number of data formats. See Chapter III for more information about data formats.

EDIT Faults

In the course of processing an EDIT instruction, the CPU performs certain checks on the data being processed. If the instruction encounters an invalid data type or number, it initiates a fault: the processor first pushes a return block onto the stack with the program counter in the return block pointing to the instruction that caused the fault. It then places a code indicating the type of fault in AC1, and executes a *Jump indirect* to the EDIT fault address, location 46_8. This location should point to a fault handling routine.

Four words of the stack are reserved for use by EDIT as temporary storage. The fault handling routine must be aware of this. When a fault occurs, the return block is pushed after these four words. The following table describes the EDIT faults and indicates the total number of words pushed on the stack as a result of the fault.

CODE	PUSHED (WORDS)	MEANING
0	9	An invalid digit or alphabetic character was encountered by DMVA or DMVN; AC2 contains SI.
1	5	Invalid data type (6 or 7); AC3 contains SI.
2	9	DMVA or DMVC op-code with source data type 5; AC2 contains SI.
3	9	An invalid op-code was encountered; AC2 contains SI.
6	5	Sign code is invalid for this data type; AC3 contains SI.

EDIT OPERATIONS[1]

Mnem	Name	Function
DADI	Add to DI	Adds an 8-bit two's complement integer, p0, to DI.
DAPS	Add to P Depending on S	If S is zero, adds two's complement integer, p0, to op-code pointer P.
DAPT	Add to P Depending on T	If T is zero, adds two's complement integer, p0, to op-code pointer P.
DAPU	Add to P	Adds two's complement integer, p0, to P.
DASI	Add to SI	Adds two's complement integer, p0, to SI.
DDTK	Decrement and Skip if Non-zero	Decrements word in stack by one; if result is non-zero adds two's complement integer, p0, to P.
DEND	End Edit	Terminates the *Edit* sub-program.
DICI	Insert Characters Immediate	Inserts j characters from the op-code stream to the destination field.
DIMC	Insert Characters J Times	Inserts the character at p0 into the destination field j times.
DINC	Insert Character Once	Inserts the character at p0 in the destination field once.
DINS	Insert Sign	If S is zero, inserts character at p0 into destination field; else inserts character at p1.
DINT	Insert Character Suppress	If T is zero, inserts character at p0 into destination field; else inserts character at p1.
DMVA	Move Alphabetics	Moves j alphabetic characters from source field to destination.
DMVC	Move Characters	Moves j characters from source field to destination field.
DMVF	Move Float	Moves j floating point numbers from source to destination fields.
DMVN	Move Numerics	Moves j numeric characters from source to destination fields.
DMVO	Move Digit with Overpunch	Adds digit from source field to p0, p1, p2, or p3, depending on digit and S, and moves result to destination field.
DMVS	Move Numeric with Zero Suppression	Moves j characters from source field to destination field replacing leading zeros and spaces with p0.
DNDF	End Float	If T and S are zero, moves p0 to destination field; if T is zero and S is one, moves p1 to destination field.
DSSO	Set S to One	Sets S to one.
DSSZ	Set S to Zero	Sets S to zero.
DSTK	Store in Stack	Stores the byte at p0 in bits 8-15 of the stack word.
DSTO	Set T to One	Sets T to one.
DSTZ	Set T to Zero	Sets T to zero.

[1] *Edit operations form a sub-program constructed of 8-bit op-codes, some of which are followed by one or more 8-bit operands.*

STACK

There are 14 stack instructions. A stack instruction:

- pushes words onto the stack;
- pops words off of the stack;
- changes the value of the stack pointer.

Most stack instructions move a single word or a return block of a given size (5 to 18 words) to or from the stack, although two instructions (*Push Multiple Accumulators* and *Pop Multiple Accumulators*) move the contents of one to four accumulators. The *Vector On Interrupting Device Code* instruction pushes up to 10 words onto the stack in some of its modes.

See the section on the stack in Chapter II for a more complete discussion of this facility.

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 instruction after the next sequential instruction 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 Chapter II.

MAP

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 the supervisor's address space.
DOC	Initiate Page Check	Identifies a logical page.
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.

DEMAND PAGING AND BREAKPOINT FACILITY

There are eight demand paging and breakpoint facility instructions in the M/600 . These instructions:

- Return the CPU to its state before the last page fault or hardware breakpoint;
- Select, read and clear the page-use flags maintained by the demand paging facility;
- Control the operation of the breakpoint facility.

All but one of the instructions are I/O instructions using the mnemonic **DPM** (device code 4). All the I/O format instructions transfer data to or from the demand paging or breakpoint facility using one of the four accumulators. The *Pop Context Block* instruction is not in I/O format.

DEMAND PAGING AND BREAKPOINT FACILITY INSTRUCTIONS

Mnem	Name	Function
DIA	Read Page-Use Flags	Loads an accumulator with the flags selected by the previous *Select Table And Word* instruction.
DIB	Read Breakpoint Control Flags	Loads bit 8-15 of an accumulator with the breakpoint control flags.
DIC	Read Breakpoint Address	Loads an accumulator with the contents of the breakpoint register.
DOA	Select Page-Use Table And Word	Selects one word of the reference or modified tables for reading or clearing.
DOB	Set Breakpoint Control Flags	Sets the breakpoint control flags according to the contents of an accumulator.
DOC	Set Breakpoint Address	Loads the breakpoint register from an accumulator.
DPOP	Pop Context Block	Returns the CPU to its state at the time of the last page fault or hardware breakpoint.
NIOC	Clear Page-Use Flags	Sets to 0 all 16 flags specified by the previous *Select Table And Word* instruction.

EXTENDED OPERATION

There are two extended operation instructions in the M/600 instruction set. These instructions:

- efficiently transfer control to another procedure;
- provide pointers to stack locations which can contain arguments;
- push a return block for efficient return from the called procedure.

The extended operation instructions can specify any two of the four accumulators, using information coded with the instruction to select the intended accumulators.

The **XOP** instruction can specify any of 32 procedure entry points, using the XOP table of addresses. The **XOP1** instruction can specify an additional 16 procedure entry points.

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.

INPUT/OUTPUT

The I/O instructions in the M/600 :

- Transfer data to and from external devices;
- Control external devices;
- Control and test flags within external devices;
- Perform special functions within the computer itself.

Most I/O instructions use a device code to specify the device which is the object of the instruction. The device code is coded with the instruction and the assembler recognizes mnemonics for most device codes (see Appendix A). All the interrupt instructions use the device code 77_8. The assembler recognizes the mnemonic **CPU** for this device code.

In addition, most I/O instructions use one of the four accumulators to hold a byte of data before it is transmitted or after it is received. The accumulator is also specified when coding the instruction.

Some of these instructions have special mnemonics which can be used in place of the standard mnemonics. The one limitation is that the mnemonics for controlling the state of flags cannot be appended to the 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.
HALTA (DOC, CPU)	Halt	Stops the Processor.
INTA (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.
IORST (DIC, CPU)	Reset	Sets all Busy and Done flags and the priority mask to 0.
MSKO (DOB, CPU)	Mask Out	Changes the priority mask.
NIO	No I/O Transfer	Changes a flag without causing any other effect.
READS (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.

BURST MULTIPLEXOR CHANNEL

Five instructions are used to program the burst multiplexor channel. These instructions:

- load and read the map table;
- read channel status;
- perform diagnostic functions.

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 software. 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.*

BASIC I/O DEVICES

The M/600 computer includes a Programmable Interval Timer, a Real Time Clock, and an Asynchronous Line Controller as basic I/O devices.

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 M/600 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.

HOST/IOP COMMUNICATION AND IOP INTERNAL FUNCTIONS

There are 10 instructions that allow Host/IOP communications or control IOP internal functions. These instructions:

- Control or monitor the operation of the IOP from the host processor;
- Control or monitor certain features of the IOP from the IOP.

Five of the instructions are used by the host processor to control and monitor the operation of the IOP. These instructions perform functions similar to those which can be performed at the console of the host by an operator (e.g., examining the contents of memory locations or accumulators, depositing values into memory locations or accumulators, stop, reset). They are I/O instructions using device code 65_8, which can be represented by the mnemonic **IOP**. (See the section on the I/O processor in Chapter II for a discussion of the cross-interrupt capability between the host and I/O processors.) These instructions use five registers in the IOP: PC save register, console buffer, address buffer, console function register, and switch register.

The PC save register is loaded by the IOP each time the IOP halts. Bits 1-15 will contain the value of the IOP program counter, and bit 0 will contain the value of the Carry bit.

(read as Busy by the IOP) allows the host to interrupt the IOP.

The console buffer corresponds to the row of Address and/or Data lights on a normal CPU. It contains the result of the last console operation performed by the IOP (such as *Examine*).

The address buffer contains the last address used in an IOP reference to Host memory, or, if the Address Save bit in the IOP map status/parity control register is set to 1, this register contains the last reference to either local or host memory. If Address Save is set when a parity error occurs, this register will contain the address of the erroneous word.

The console function register takes the place of the function keys on a normal CPU. The host loads the appropriate bit pattern into this register to perform each function.

The switch register corresponds to the row of toggle switches on a normal CPU front panel. It contains the data or address for the IOP's console functions.

The host and IOP can also communicate via a cross/interrupt facility. There are four cross/interrupt status flags, an IOP Done flag, Host Busy flag, and IOP Run flag. The IOP Run flag (read as Busy by the Host computer) is 1 if the IOP is running and is 0 if it is stopped. The IOP Done flag (read as Done by the host processor) allows the IOP to interrupt the host processor. The Host Busy flag

HOST/IOP COMMUNICATION INSTRUCTIONS

Mnem	Name	Function
DIA	Read PC Save Register	Loads the contents of the IOP PC save register into an accumulator.
DIB	Read Console Buffer	Loads the contents of the IOP console buffer into an accumulator.
DIC	Read Address Buffer	Loads the contents of the IOP address buffer into an accumulator.
DOA	Control Console Function Register	Stores the contents of an accumulator into the IOP console function register.
DOB	Control Switch Register	Stores the contents of an accumulator into the IOP switch register.

Five of the IOP-related instructions are used by the IOP to control and monitor its MAP, timer, or micro-interrupt facility. Four of these are I/O instructions which use the device code 4, or the mnemonic **IOPI**. The fifth is the *IOP Load Map* instruction.

IOP INSTRUCTIONS

Mnem	Name	Function
DIA	IOP Read Map Status	Loads the map status and parity control bits and MAP host/local flag into an accumulator.
DOA	IOP Control Map And Parity	Stores the contents of an accumulator into the map status and parity control register.
DOB	IOP Generate Micro-interrupt	Causes execution of a host *Load Map* instruction which is invisible to the host program.
DOC	IOP Control Timer	Stores the contents of an accumulator into the timer control register.
LMP	IOP Load Map	Loads a number of map entries from a table in memory to the IOP MAP.

HOST/DCU COMMUNICATION AND DCU INTERNAL FUNCTIONS

There are eight instructions that support Host/DCU communications or control the DCU internal functions. These instructions:

- Control or monitor the operation of the DCU from the Host processor;
- Control or monitor the operation of the DCU Real Time Clock;

Five of the instructions are used by the host processor to control and monitor the operation of the DCU. These instructions perform functions similiar to those which can be performed at the console of the host by an operator (e.g., examining the contents of memory locations or accumulators, depositing values into memory locations or accumulators, stop, reset). These I/O instructions use device code $34_8$for the first DCU in a system. This device code is represented by the mnemonic DCU. The host and DCU use three registers to communicate; the Host To Data Control Unit (HTDCU) register, the DCU command register and the Diagnostic Data Register. Additional communication is provided by a cross interrupt facility.

The HTDCU register is a communications link from the host computer to the DCU. The host computer can place data in this register and the data control unit can read it.

The DCU command register provides the host processor with control over the operation of the DCU. Through this register, the host processor can start, stop, reset, and place the DCU in diagnostic mode.

There are three Cross-Interrupt/Status flags in the Host/DCU interface: the DCU Run flag, the DCU Done flag, and the Host Done flag. The DCU Run flag (read as Busy by the host computer) is 1 if the DCU is running and 0 if the DCU is stopped. The DCU Done flag (read as Done by the host processor) allows the DCU to interrupt the host processor. The Host Done flag (read as Done by the DCU) allows the host processor to interrupt the DCU.

HOST TO DCU INSTRUCTIONS

Mnem	Name	Function
DOA	Control Mode	Used to issue commands to the DCU which can start the DCU at a specified address, stop the DCU, or continue the DCU from where it had stopped. Also used by diagnostics.
DOB	Load HTDCU Register	Places the contents of an accumulator in the Host To DCU register.
DIA	Read Diagnostic	Loads Diagnostic information from the DCU into an accumulator.
DIB	Read Program Counter	Loads the contents of the DCU's Program Counter into an accumulator.
DIC	Reset	Stops the DCU, places it in diagostic mode, and forces an I/O reset to all DCU I/O devices.

The DCU uses three instructions to communicate with the Host, control its real time clock, and modify the data channel map selection register. These instructions use device code 76_8Which is represented by the mnemonic DCUI. These instructions are shown in the table below.

DCU INSTRUCTIONS

Mnem	Name	Function
DIA	Read HTDCU	Loads the contents of the Host to DCU register into an accumulator.
DOA	Control Real Time Clock	Used to turn the DCU real time clock on or off.
DOC	Select Data Channel Map	Selects which data channel map will be used to map DCU addresses into host memory.

POWER FAIL

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.

ERROR CHECK AND CORRECTION

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.

Chapter V
M/600 INSTRUCTIONS

Chapter V lists all the instructions for the M/600 *except* those I/O instructions intended for a specific device such as the MAP, the BMC, and special CPU instructions. 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

Some instructions can only be executed by the host processor, while others can also be executed by the I/O processor and/or the Data Control Unit. A label with each instruction indicates which processors can execute that 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., *[,skipl)* 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:

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

ASSEMBLER CONVENTIONS

Some of the more important relationships between assembler and machine instructions are discussed below. For a detailed discussion of the assembler, see the assembler manual listed in the Bibliography.

Location Labels

Any word of your program may be labelled for easy referencing. Label a word by coding a label followed by a colon before the word, e.g., **START: LDA O,DATA.** Here, **START** is a label for this word and **DATA** is a reference to another word with the label **DATA.**

Instruction Fields

Most instruction fields can be coded as expressions combining numbers, symbols and arithmetic/logic operators, e.g., **ELDA** *ac*,**TABLE+SIZE,3.**

Pseudo-Ops

The assembler expects to find certain expressions in the user program which control the operation of the assembler, but do not directly control the processing of the program after assembly. These expressions are called *pseudo-ops.* An example of a pseudo-op is **.END** which marks the end of a source program.

Lower Page Zero

Extended class memory reference instructions can address any one of 32,768 words (i.e., all of the maximum logical address space) in memory using absolute addressing (see Chapter III). The short class of instruction, however, has only 8 bits in the displacement field, and therefore can only address locations $0\text{-}377_8$ ($0\text{-}255_{10}$) using absolute addressing. The first 256 words of memory are therefore given a special status, because they are accessible by any memory reference instruction anywhere in the program, using absolute addressing. The assembler manual refers to these first 256 locations as *lower page zero.*

Radix

The assembler assumes that numbers in the instruction fields are in the numbering system corresponding to the current *radix.* The default radix is base 8, or octal. Decimal, hexadecimal and binary are other commonly used radices. The assembler radix can be changed using the **.RDX** pseudo-op.

Effective Addresses

In the instructions which calculate an effective address, the assembler expects the following coding conventions to be used:

Coding the symbol @ anywhere in the effective address operand string sets the indirect bit to 1.

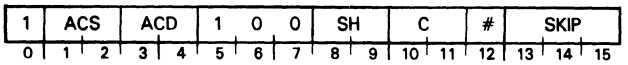
Coding a comma followed by one of digits 0-3 as the last operand of the operand string sets that value in the index field. Use the symbol dot *(.) to set the index bits to 01.* Dot *can be read to mean* address of the instruction. *When the dot is used, it is followed by either a plus or minus sign followed by the displacement, e.g.,* **.+7,** *or* **.-2.**

When coding extended class instructions, setting the index bits to 01 using the dot (e.g., **EJMP .+d***) does not produce the same effect as coding a comma followed by a 1 (***EJMP d,1***). When using the dot, the displacement is added to the address of the instruction (the first word of a 2-word instruction). When using the comma, the displacement is added to the address of the word containing the displacement (the second word of a 2-word instruction). Therefore,* **EJMP .+3** *is equivalent to* **EJMP 2,1.**

Add Complement

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

Valid for: CPU - IOP - DCU



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

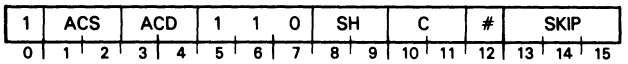
Initializes the carry bit to the specified value, adds the logical complement of the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The instruction then performs the specified shift operation, and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE: *If the number in ACS is less than the number in ACD, the instruction complements the Carry bit.*

Add

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

Valid for: CPU - IOP - DCU



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

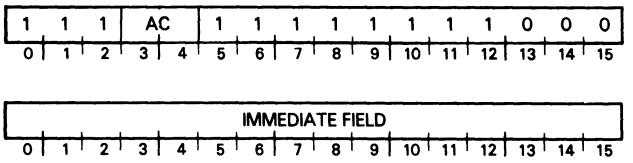
Initializes the carry bit to the specified value, adds the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD, and places the result in the shifter. If the addition produces a carry of 1 out of the high-order bit, the carry bit is complemented. The instruction then performs the specified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

NOTE: *If the sum of the two numbers being added is greater than 65,535, the instruction complements the Carry bit.*

Extended Add Immediate

ADDI *i,ac*

Valid for: CPU - IOP



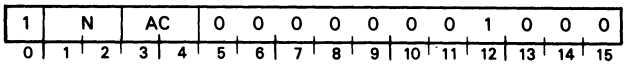
Adds a signed integer in the range -32,768 to +32,767 to the contents of an accumulator.

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. The Carry bit remains unchanged.

Add Immediate

ADI *n,ac*

Valid for: CPU - IOP



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. The carry bit remains unchanged.

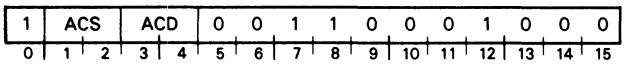
NOTE: *The assembler takes the coded value of* **n** *and subtracts one from it before placing it in the immediate field. Therefore, you should code the exact value that you wish to add.*

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

AND With Complemented Source

ANC *acs,acd*

Valid for: CPU - IOP

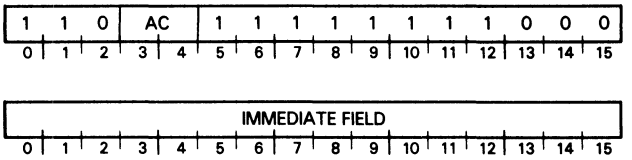


Forms the logical AND of the logical complement of the contents of ACS and the contents of ACD and places the result in ACD. The instruction sets a bit position in the result to 1 if the corresponding bit positions in ACS and ACD contain a 0 and a 1, respectively. The contents of ACS remain unchanged.

AND Immediate

ANDI *i,ac*

Valid for: CPU - IOP

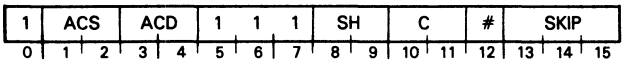


Places the logical AND of the contents of the immediate field and the contents of the specified accumulator in the specified accumulator.

AND

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

Valid for: CPU - IOP - DCU



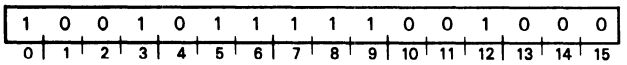
Forms the logical AND of the contents of two accumulators.

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.

Block Add and Move

BAM

Valid for: CPU - IOP



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.

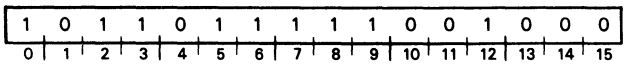
> NOTE: *Because of the potentially long time that may be required to perform this instruction it is interruptable. If a* Block Add and Move *instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the interrupted instruction. Because 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

Valid for: CPU - IOP



Moves memory words from one location to another.

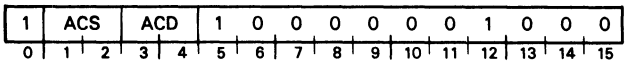
The *Block Move* instruction is the same as the *Block Add And Move* instruction in all respects except that no addition is performed and AC0 is not used.

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

Set Bit To One

BTO *acs,acd*

Valid for: CPU



Sets the specified bit to 1.

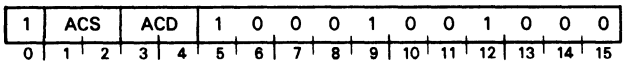
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16-bits of the bit pointer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

Set Bit To Zero

BTZ *acs,acd*

Valid for: CPU



Sets the addressed bit to 0.

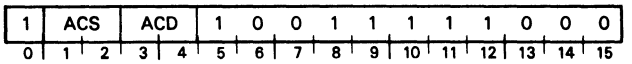
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer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

Compare To Limits

CLM *acs,acd*

Valid for: CPU - IOP



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.

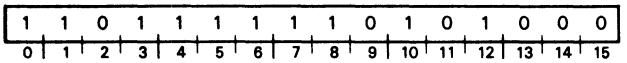
If ACS and ACD are specified as different accumulators, the address of the limit value L is contained in bits 1-15 of ACD. The limit value H is contained in the word following L. Bit 0 of ACD is ignored.

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

Valid for: CPU - IOP



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

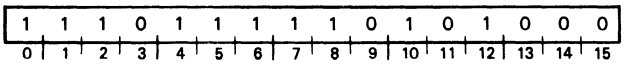
The strings may overlap in any way. Overlap will not effect the results of the comparison.

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 was found), or to the byte following string 2 (if string 2 was exhausted). AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality was found), or to the byte following string 1 (if string 1 was exhausted). If the length of both string 1 and string 2 was zero, the instruction returns 0 in AC1. If the two strings are of unequal length, the instruction *fakes* space characters $<040_8>$ in place of bytes from the exhausted string, and continues the comparison.

Character Move Until True

CMT

Valid for: CPU - IOP



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.

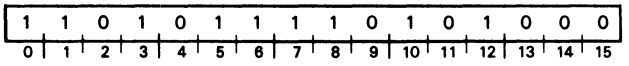
Upon completion, AC0 contains the resolved address of the translation table and AC1 contains the number of bytes that were not moved. AC2 contains a byte pointer to the byte following the last byte written in the destination field. AC3 contains a byte pointer either to the delimiter or to the first byte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

Valid for: CPU - IOP



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.

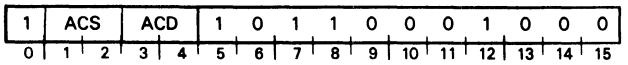
> NOTE: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters.*

If the source field is shorter than the destination field, the instruction pads the destination field with space characters $<40_8>$. If the source field is longer than the destination field, the instruction terminates when the destination field is filled and returns the value 1 in the Carry bit, otherwise the instruction returns the value 0 in the Carry bit.

Count Bits

COB *acs,acd*

Valid for: CPU - IOP



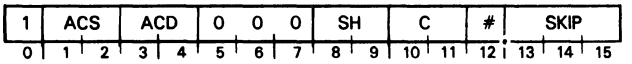
Adds a number equal to the number of ones in ACS to the signed, 16-bit, two's complement number in ACD. The instruction leaves the contents of ACS and the state of the carry bit unchanged.

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]*

Valid for: CPU - IOP - DCU



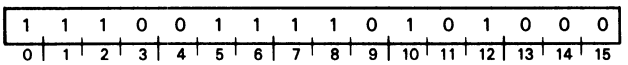
Forms the logical complement of the contents of an accumulator.

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

Character Translate

CTR

Valid for: CPU - IOP



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 3.

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

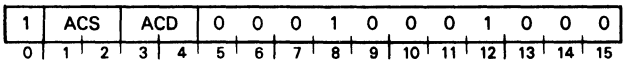
If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

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.

Decimal Add

DAD *acs,acd*

Valid for: CPU - IOP



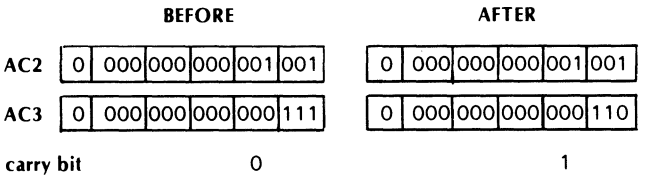
Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses the carry bit for a decimal carry.

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:

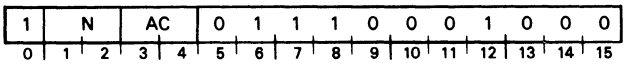
Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0. After the instruction **DAD 2,3** is executed, AC2 remains the same; bits 12-15 of AC3 contain 6; and the carry bit is 1, indicating a decimal carry from this *Decimal Add.*



Double Hex Shift Left

DHXL *n,ac*

Valid for: CPU - IOP



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.*

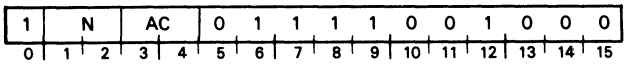
The assembler takes the coded value of n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

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*

Valid for: CPU - IOP



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.*

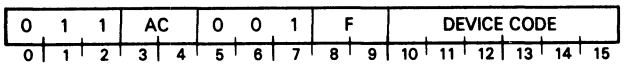
The assembler takes the coded value of n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

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

Data In A

DIA *[f] ac,device*

Valid for: CPU - IOP - DCU



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

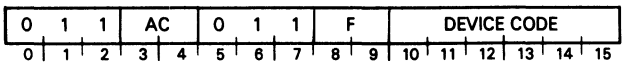
The contents of the A input buffer in the specified device are placed in the specified AC. After the data transfer, the Busy and Done flags are set according to the function specified by F.

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 B

DIB *[f] ac,device*

Valid for: CPU - IOP - DCU



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

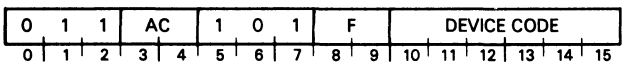
Places the contents of the B input buffer in the specified device in the specified AC. After the data transfer, sets the Busy and Done flags according to the function specified by F.

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 C

DIC *[f] ac,device*

Valid for: CPU - IOP - DCU



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

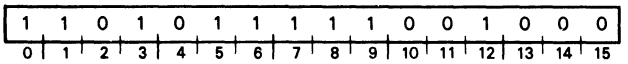
Places the contents of the C input buffer in the specified device in the specified AC. After the data transfer, sets the Busy and Done flags according to the specified F.

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.

Unsigned Divide

DIV

Valid for: CPU - IOP



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

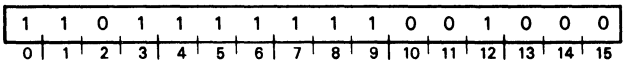
Divides the unsigned 32-bit number contained in AC0 and AC1 by the unsigned, 16-bit number in AC2. The quotient and remainder are unsigned, 16-bit numbers and are placed in AC1 and AC0, respectively. The carry bit is set to 0. The contents of AC2 remain unchanged.

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

Signed Divide

DIVS

Valid for: CPU - IOP



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

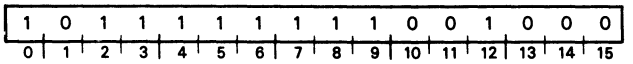
The signed, 32-bit two's complement number contained in AC0 and AC1 is divided by the signed, 16-bit two's complement number in AC2. The quotient and remainder are signed, 16-bit numbers and occupy AC1 and AC0, respectively. The sign of the quotient is determined by the rules of algebra. The sign of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

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

Sign Extend and Divide

DIVX

Valid for: CPU - IOP



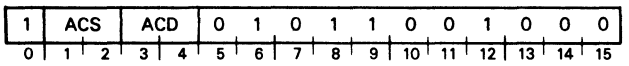
Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.

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*

Valid for: CPU - IOP



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.

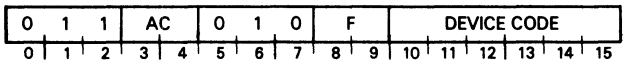
AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 8-15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. The Carry bit and the contents of ACS remain unchanged.

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

Data Out A

DOA*[f] ac,device*

Valid for: CPU - IOP - DCU



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

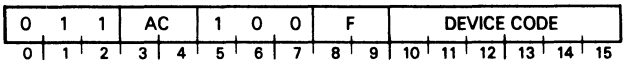
Places the contents of the specified AC in the A output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by F. The contents of the specified AC remain unchanged.

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*

Valid for: CPU - IOP - DCU



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

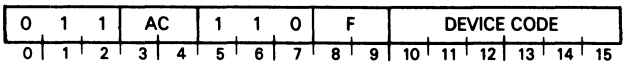
Places the contents of the specified AC in the B output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by F. The contents of the specified AC remain unchanged.

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*

Valid for: CPU - IOP - DCU



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

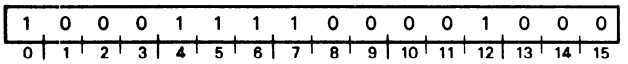
Places the contents of the specified AC in the C output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by F. The contents of the specified AC remain unchanged.

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

Pop Context Block

DPOP

Valid for: CPU



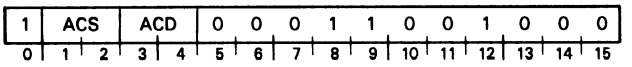
Uses the information in the context block pointed to by location 10_8 to restore the CPU state to that at the time of the last page fault or hardware breakpoint. If bit 1 in the status word (offset 8 of the block) is 0, this indicates that the fault occured in **MAP A**, and the instruction restores the floating point unit state from the top 18 words of the block. Execution of the interrupted program resumes before, during, or after the instruction which caused the fault, depending on the instruction type and how far it had proceeded before the fault.

> **NOTE: DPOP** *is a privileged instruction which can execute only in Map B's address space; and the context block pointer (location 10_8 must be in that space). Issuing the instruction from Map A's address space results in an I/O protection fault* whether or not I/O protection is specified for that map. *The result of executing this instruction in unmapped address space is undefined.*

Decimal Subtract

DSB *acs,acd*

Valid for: CPU - IOP

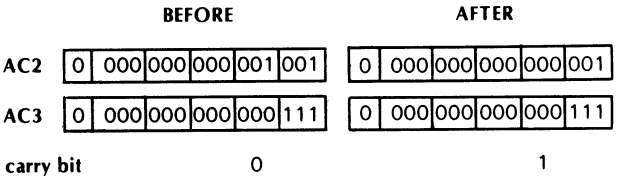


Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses the carry bit as a decimal borrow.

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 the carry bit 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 the carry bit. In other words, if the final result is negative, the instruction indicates a borrow and sets the carry bit to 0. If the final result is positive, the instruction indicates no borrow and sets the carry bit to 1. The contents of ACS and bits 0-11 of ACD remain unchanged.

Example:

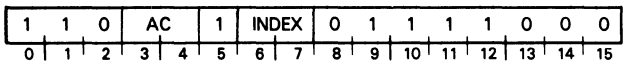
Assume that bits 12-15 of AC2 contain 9; bits 12-15 of AC3 contain 7; and the carry bit is 0. After the instruction **DSB 3,2** is executed, AC3 remains the same; bits 12-15 of AC2 contain 1; and the carry bit is set to 1, indicating no borrow from this *Decimal Subtract.*

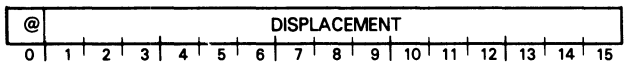


Dispatch

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

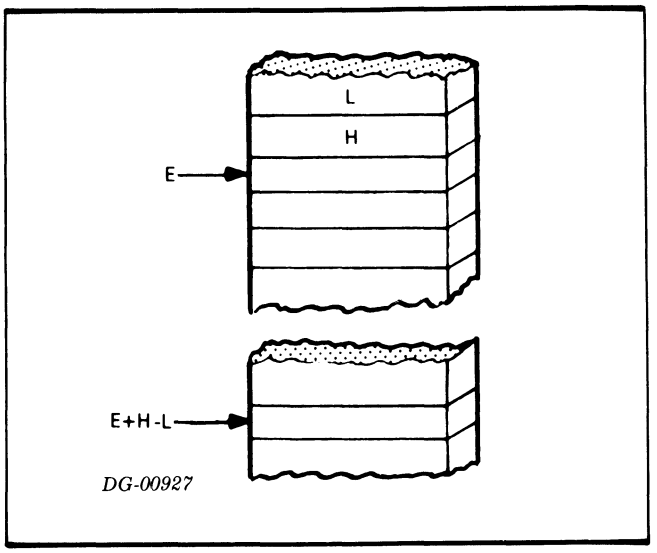
Valid for: CPU - IOP





Conditionally transfers control to an addres 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.



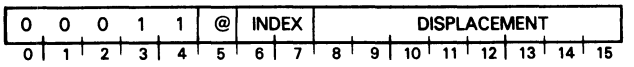
Compares the signed, two's complement number contained in the accumulator to the limit words. If the number in the accumulator is less than *L* or greater than *H*, sequential operation continues with the instruction immediately after the *Dispatch* instruction.

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]*

Valid for: CPU - IOP - DCU



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.

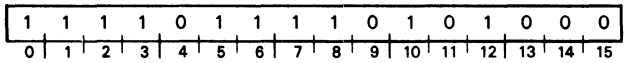
NOTE: *Issued to the CPU, this instruction both reads and possibly modifies the contents of a memory location in a single memory operation. Issued by an IOP or a DCU, the instruction may read and modify memory in separate memory operations if the effective address* E *lies outside that processor's local physical memory space. Each such separate memory operation propagates to host memory via the host data channel. All logical addresses above* 1777_8 *in the DCU, and memory references mapped to the host from the IOP lie outside local physical memory.*

Contention on the data channel, or activity on the burst multiplexor channel may interleave the two data channel transfers. It is possible for the interleaving operation to alter one memory location in the time between the read and the modify cycles of the remotely-issued instruction.

Edit

EDIT

Valid for: - CPU



Converts a decimal number from either packed or unpacked form to a string of bytes under the control of an edit sub-program. This sub-program can perform many different operations on the number and its destination field including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. The instruction also performs operations on alphanumeric data if data type 4 is specified.

The instruction maintains two flags and three indicators or pointers.

The flags are the significance Trigger (T) and the Sign flag (S). T is set to 1 when the first non-zero digit is processed unless otherwise specified by an edit op-code. At the beginning of an *Edit* instruction, T is set to 0. S is set to reflect the sign of the number being processed. If the number is positive, S is set to 0. If the number is negative, S is set to 1.

The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the op-code Pointer (P). Each is 16 bits wide and contains a byte pointer to the *current* byte in each respective area. At the beginning of an *Edit* instruction, SI is set to the value contained in AC3. DI is set to the value contained in AC2, and P is set to the value contained in AC0. Also at this time the sign of the source number is checked for validity.

The sub-program is made up of 8-bit op-codes followed by one or more 8-bit operands. P, a byte pointer, acts as the *program counter* for the *Edit* sub-program. The sub-program proceeds sequentially until a branching operation occurs - much the same way programs are processed. Unless instructed to do otherwise, the *Edit* instruction updates P after each operation to point to the next sequential op-code. The instruction continues to process 8-bit op-codes until directed to stop by the DEND op-code.

The sub-program can test and modify S and T, as well as modify SI, DI and P.

Upon entry to EDIT AC0 is a byte pointer to the first op-code of the *Edit* sub-program.

AC1 is the data-type indicator describing the number to be processed.

AC2 is a byte pointer to the the first byte of the destination field.

AC3 is a byte pointer to the first byte of the source field.

The fields may overlap in any way. However the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

Upon successful termination, the carry bit contains the significance Trigger; AC0 contains a byte pointer (P) to the next op-code to be processed; AC1 is undefined; AC2 contains a byte pointer (DI) to the next destination byte; and AC3 contains a byte pointer (SI) to the next source byte.

NOTES: *If SI is moved outside the area occupied by the source number, zeros will be supplied for numeric moves, even if SI is later moved back inside the source area.*

Some op-codes perform movement of characters from one string to another. For those op-codes which move numeric data, special actions may be performed. For those which move non-numeric data, characters are copied exactly to the destination.

The Edit *instruction places information on the stack. Therefore, the stack must be set up and have at least 9 words available for use.*

If the Edit *instruction is interrupted, it places restart information on the stack and places* 177777_8 *in AC0.*

If the initial contents of AC0 are equal to 177777_8 *the* Edit *instruction assumes it is restarting from an interrupt; therefore do not allow this to occur under any other circumstances.*

In the description of some of the *Edit* op-codes we use the symbol *j* to denote how many characters a certain operation should process. When the high order bit of *j* is 1, *j* has a different meaning, it is a pointer into the stack to a word that denotes the number of characters the instruction should process. So, in those cases where the high order bit of *j* is 1, the instructions interpret *j* as an 8-bit two's complement number pointing into the stack. The number on the stack is at address:

stack pointer + 1 + *j*.

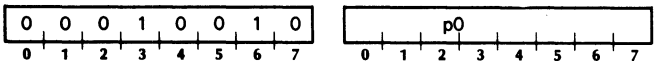
The operation uses the number at this address as a character count instead of *j*.

An *Edit* operation that processes numeric data (e.g., DMVN) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

Add To DI

DADI *p0*

Valid for: - CPU

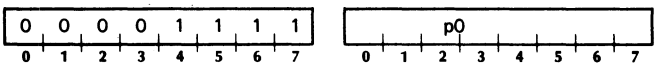


Adds the 8-bit two's complement integer specified by *p0* to the Destination Indicator (DI).

Add To P Depending On S

DAPS *p0*

Valid for: - CPU

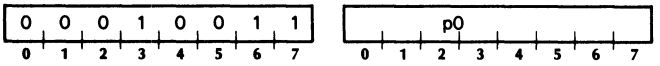


If S is 0, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPS** op-code.

Add To SI

DASI *p0*

Valid for: - CPU

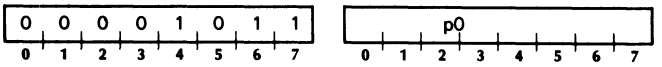


Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

Add To P Depending On T

DAPT *p0*

Valid for: - CPU

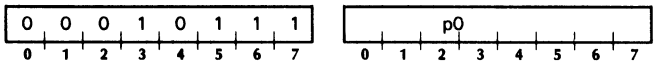


If T is one, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPT** op-code.

Add To P

DAPU *p0*

Valid for: - CPU

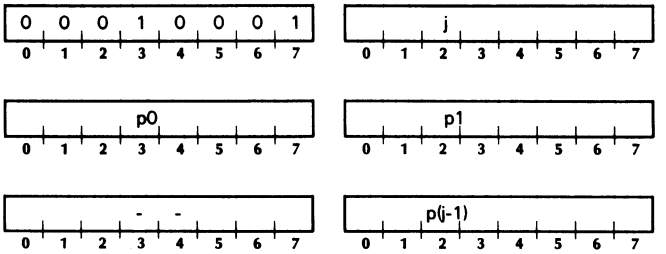


Adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPU** op-code.

Insert Characters Immediate

DICI *j,p0,p1,...,p(j-1)*

Valid for: -CPU

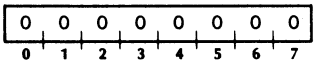


Inserts *j* characters from the op-code stream into the destination field beginning at the position specified by DI. Increases P by (*j*+2), and increases DI by *j*.

End Edit

DEND

Valid for: - CPU

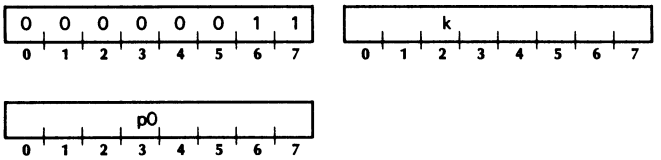


Terminates the EDIT sub-program.

Decrement and Jump If Non-Zero

DDTK *k,p0*

Valid for: - CPU

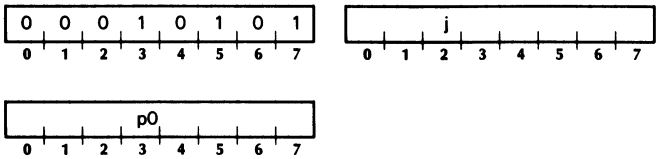


Decrements a word in the stack by one. If the decremented value of the word is non-zero, the instruction adds the 8-bit two's complement integer specifed by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DDTK** op-code. If the 8-bit two's complement integer specified by *k* is negative, the word decremented is at the address (stack pointer+1+*k*). If *k* is positive, the word decremented is at the address (frame pointer+1+*k*).

Insert Character J Times

DIMC *j,p0*

Valid for: - CPU

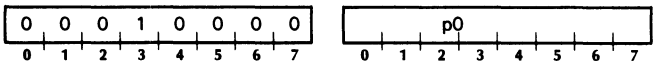


Inserts the character specified by *p0* into the destination field a number of times equal to *j* beginning at the position specified by DI. Increases DI by *j*.

Insert Character Once

DINC *p0*

Valid for: - CPU

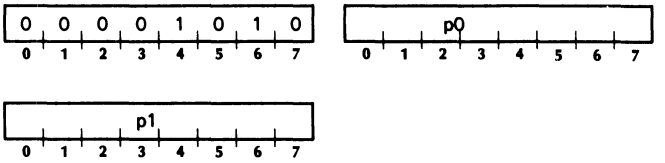


Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by 1.

Insert Character Suppress

DINT *p0,p1*

Valid for: - CPU

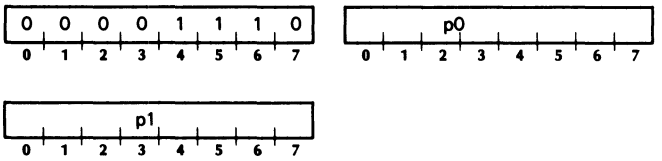


If the significance Trigger (T) is 0, the instruction inserts the character specifed by *p0* in the destination field at the position specified by DI. If T is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Insert Sign

DINS *p0,p1*

Valid for: - CPU

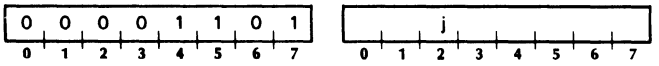


If the Sign flag (S) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If S is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Move Alphabetics

DMVA *j*

Valid for: - CPU



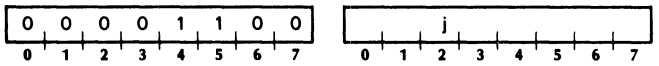
Moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets T to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source field is data type 5 (packed). Initiates a commercial fault if any of the characters moved is not an alphabetic (A-Z, a-z, or space).

Move Characters

DMVC *j*

Valid for: - CPU



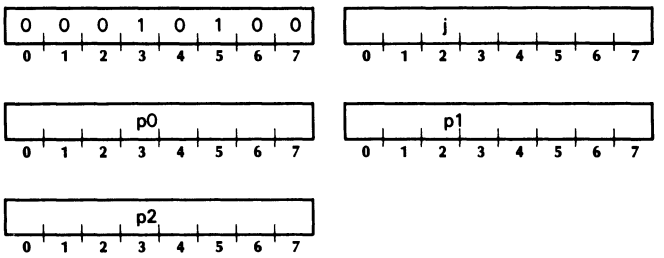
Increments SI if the source data type is 3 and *j*>0. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets T to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Performs no validation of the characters.

Move Float

DMVF *j,p0,p1,p2*

Valid for: - CPU



If the source data type is 3, j>0, and SI points to the sign of the source number, the instruction increments SI. Then for *j* characters, the instruction either places a digit substitute in the destination field beginning at the position specified by DI, or it moves a digit from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. When T changes from 0 to 1, the instruction places both the digit substitute and the digit in the destination field, and increases SI by *j*. If T does not change from 0 to 1, the instruction increases DI by *j*. If T does change from 0 to 1, the instruction increases DI by *j*+1.

If the source data type is 2, the state of SI is undefined after the least significant digit has been processed.

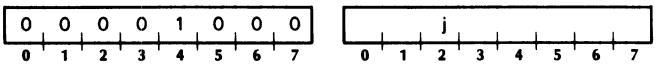
If T is 1, the instruction moves each digit processed from the source field to the destination field. If T is 0 and the digit is a zero or space, the instruction places *p0* in the destination field. If T is 0 and the digit is a non-zero, the instruction sets T to 1 and the characters placed in the destination field depend on S. If S is 0, the instruction places *p1* in the destination field followed by the digit. If S is 1, the instruction places *p2* in the destination field followed by the digit.

The instruction initiates a commercial fault if any of the digits processed is not valid for the specified data type.

Move Numerics

DMVN *j*

Valid for: - CPU



Increments SI if the source data type is 3 and $j>0$. The instruction then moves j characters from the source field beginning at the position specified by SI to the destination field beginning at the position specifed by DI. Increases both SI and DI by j. Sets T to 1.

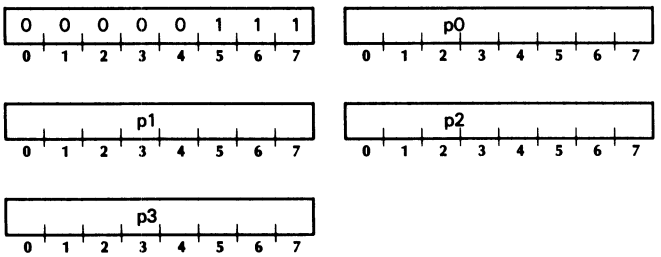
Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

For data type 2, the state of SI is undefined after the least significant digit has been processed.

Move Digit With Overpunch

DMVO *p0,p1,p2,p3*

Valid for: - CPU



Increments SI if the source data type is 3 and SI points to the sign of the source number. The instruction then either places a digit substitute in the destination field at the position specified by DI,or it moves a digit plus overpunch the source field at the position specified by SI to the destination field at the position specified by DI. Increases both SI and DI by 1.

If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.

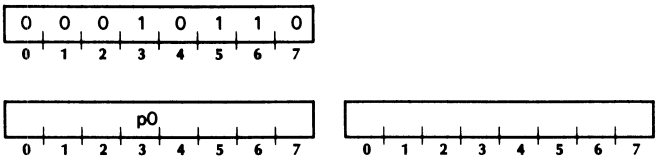
If the digit is a zero or space and S is 0, then the instruction places *p0* in the destination field. If the digit is a zero or space and S is 1, then the instruction places *p1* in the destination field. If the digit is a non-zero and S is 0, the instruction adds *p2* to the digit and places the result in the destination field. If the digit is a non-zero and S is 1, the instruction adds *p3* to the digit and places the result in the destination field. If the digit is a non-zero the instruction sets T to 1. The instructions assumes *p2* and *p3* are ASCII characters.

The instruction initiates a commercial fault if the character is not valid for the specified data type.

Move Numeric With Zero Suppression

DMVS *j,p0*

Valid for: - CPU



Increments SI if the source data type is 3, $j>0$, and SI points to the sign of the source number. The instruction then moves j characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Moves the digit from the source to the distination if T is 1. Replaces all zeros and spaces with *p0* as long as T is 0. Sets T to 1 when the first non-zero digit is encountered. Increases both SI and DI by *j*.

If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.

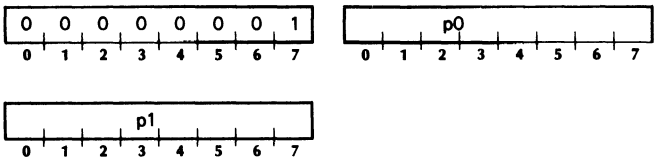
This op-code destroys the attribute specifier word.

Initiates a commercial fault if any of the characters moved is not a numeric (0-9 or space).

End Float

DNDF *p0,p1*

Valid for: - CPU

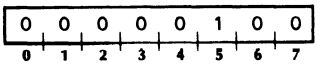


If T is 1, the instruction places nothing in the destination field and leaves DI unchanged. If T is 0 and S is 0, the instruction places *p0* in the destination field at the position specified by DI. If T is 0 and S is 1, the instruction places *p1* in the destination field at the position specified by DI. Increases DI by 1, and sets T to one.

Set S To Zero

DSSZ

Valid for: - CPU

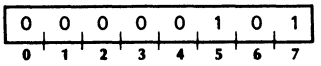


Sets the Sign flag (S) to 0.

Set S To One

DSSO

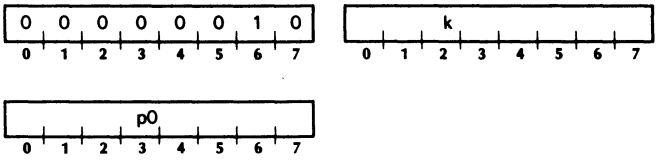
Valid for: - CPU



Sets the Sign flag (S) to 1.

Store In Stack

DSTK *k,p0*

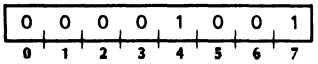


Stores the byte specified by *p0* in bits 8-15 of a word in the stack. Sets bits 0-7 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by *k* is negative, the instruction addresses the word receiving *p0* by (stack pointer+1+*k*). If *k* is positive then the instruction stores *p0* at the address (frame pointer+1+*k*).

Set T To One

DSTO

Valid for: - CPU

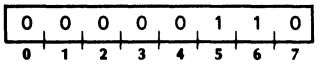


Sets the significance Trigger (T) to 1.

Set T To Zero

DSTZ

Valid for: - CPU

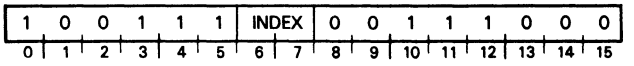


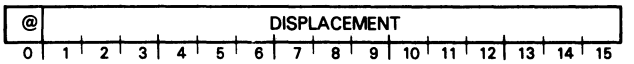
Sets the significance Trigger (T) to 0.

Extended Decrement and Skip if Zero

EDSZ ,*[@]displacement[,index]*

Valid for: CPU - IOP





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 word is zero, the instruction skips the next sequential word.

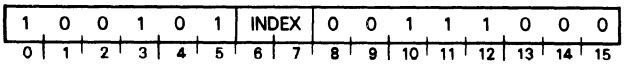
NOTE: *Issued to the CPU, this instruction both reads and possibly modifies the contents of a memory location in a single memory operation. Issued by an IOP or a DCU, the instruction may read and modify memory in separate memory operations if the effective address* E *lies outside that processor's local physical memory space. Each such separate memory operation propagates to host memory via the host data channel. All logical addresses above* 1777_8 *in the DCU, and memory references mapped to the host from the IOP lie outside local physical memory.*

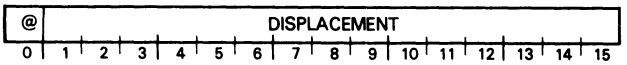
Contention on the data channel, or activity on the burst multiplexor channel may interleave the two data channel transfers. It is possible for the interleaving operation to alter one memory location in the time between the read and the modify cycles of the remotely-issued instruction.

Extended Increment And Skip If Zero

EISZ *[@]displacement[,index]*

Valid for: CPU - IOP





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

Computes the effective address *E* , and increments the contents that of memory location by one and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction increments the program counter by one and continues sequential operation at the updated value of the program counter.

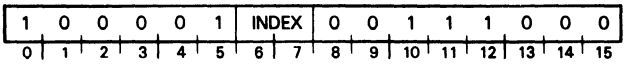
NOTE: *Issued to the CPU, this instruction both reads and possibly modifies the contents of a memory location in a single memory operation. Issued by an IOP or a DCU, the instruction may read and modify memory in separate memory operations if the effective address* E *lies outside that processor's local physical memory space. Each such separate memory operation propagates to host memory via the host data channel. All logical addresses above* 1777_8 *in the DCU, and memory references mapped to the host from the IOP lie outside local physical memory.*

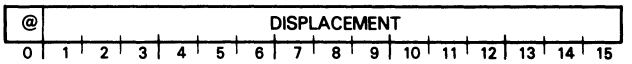
Contention on the data channel, or activity on the burst multiplexor channel may interleave the two data channel transfers. It is possible for the interleaving operation to alter one memory location in the time between the read and the modify cycles of the remotely-issued instruction.

Extended Jump

EJMP *[@]displacement[,index]*

Valid for: CPU - IOP



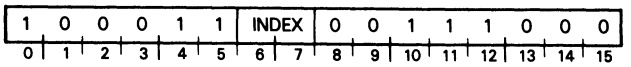


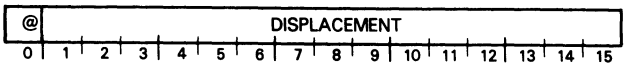
Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Extended Jump To Subroutine

EJSR *[@]displacement[,index]*

Valid for: CPU - IOP





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

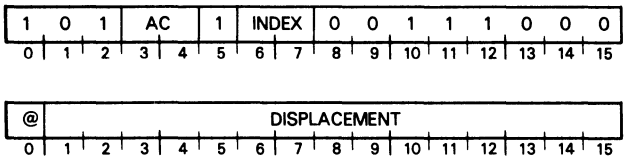
Computes the effective address, *E*; then places the address of the next sequential instruction in AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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

Extended Load Accumulator

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

Valid for: CPU - IOP



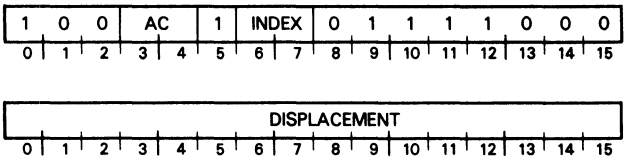
Moves a word out of memory and into an accumulator.

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

Extended Load Byte

ELDB *ac,displacement[,index]*

Valid for: CPU - IOP



Copies a byte from memory into an accumulator.

Forms a byte pointer by first taking an index value, multiplying it by 2, and then adding the low-order 16 bits of the result to the displacement. Copies the byte addressed by this byte pointer into bits 8-15 of the specified accumulator, and sets bits 0-7 of that 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.

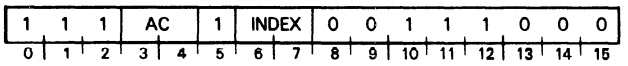
The argument *index* selects the source of the index value. It may have values in the range of 0-3; the meaning of each value is shown below:

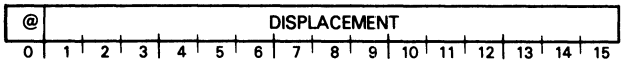
INDEX BITS	INDEX VALUE
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Load Effective Address

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

Valid for: CPU - IOP





Places an effective address in an accumulator.

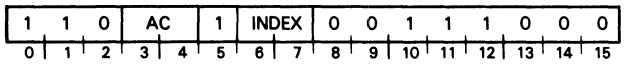
Computes the effective address, E, and places it in bits 1-15 of the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the accumulator are lost.

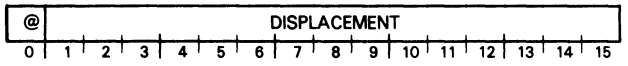
```
ELEF   0,TABLE     ; The logical address of TABLE
                   ; is placed in AC0.
ELEF   1,-55,3     ; Subtracts 000055 (octal) from
                   ; the unsigned integer in AC3 and
                   ; the result is placed in AC1.
ELEF   0,.+0       ; Places the logical address of this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

Extended Store Accumulator

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

Valid for: CPU - IOP





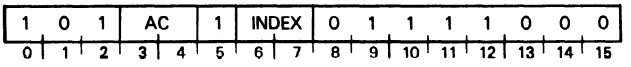
Stores the contents of an accumulator into a memory location.

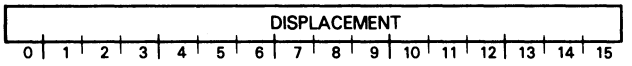
The contents of the specified accumulator are placed in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator remain unchanged.

Extended Store Byte

ESTB *ac,displacement[,index]*

Valid for: CPU - IOP





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

Forms a byte pointer by first taking an index value, multiplying it by 2, and then adding the low-order 16 bits of that result to the displacement. Copies bits 8-15 of the specified accumulator into memory at the byte address specified by the computed byte pointer. The instruction does not alter the specified accumulator.

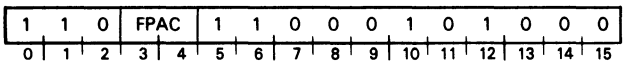
The argument *index* selects the source of the index value. It may have values in the range of 0-3; the meaning of each value is shown below:

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*

Valid for: CPU

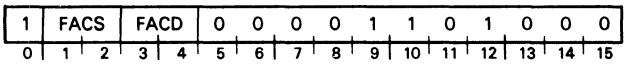


Sets the sign bit of FPAC to 0. Also sets the exponent to zero if the mantissa is zero; otherwise leaves bits 1-63 of FPAC unchanged. Updates the Z and N flags in the floating point status register to reflect the new contents of FPAC.

Add Double (FPAC to FPAC)

FAD *facs,facd*

Valid for: CPU



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.

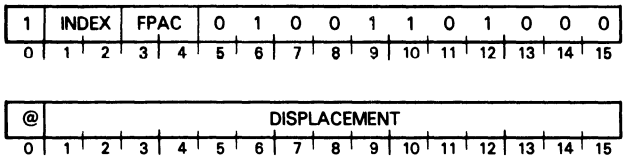
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FACD and the instruction terminates.

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]*

Valid for: CPU



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.

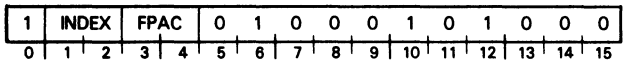
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FPAC and the instruction terminates.

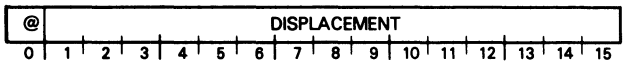
If the mantissa is non-zero, the intermediate result is normalized, and the number placed in the FPAC. If the normalization results in an exponent underflow, the UNF bit is set in the floating point status register and the instruction is terminated. The number in the FPAC is correct except that the exponent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

Add Single (Memory to FPAC)

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

Valid for: CPU





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.

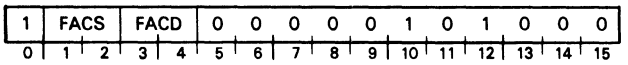
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FPAC and the instruction terminates.

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*

Valid for: CPU



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.

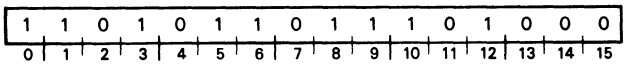
If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FACD and the instruction is terminated.

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

Valid for: CPU



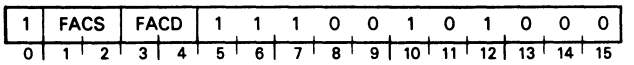
Sets bits 0-4 of the floating point status register to 0.

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

Compare Floating Point

FCMP *facs,facd*

Valid for: CPU



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.

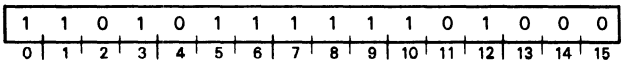
Z	N	RESULT
1	0	FACS=FACD
0	1	FACS>FACD
0	0	FACS<FACD

NOTE: *Unnormalized operands give unspecified results.*

Cosine Double

FCOSD

Valid for: CPU



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.

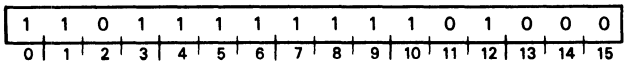
Format: Algorithm coefficients must follow the *Cosine* instruction. The format is:

RELATIVE 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

Valid for: CPU



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.

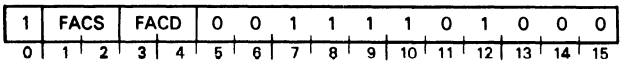
Format: Algorithm coefficients must follow the *Cosine* instruction. The format is:

RELATIVE 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*

Valid for: CPU



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.

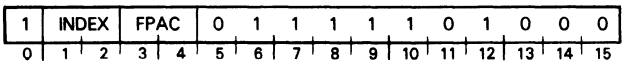
The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The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

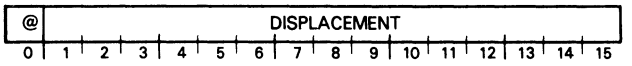
If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FACD is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Divide Double (FPAC by Memory)

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

Valid for: CPU





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.

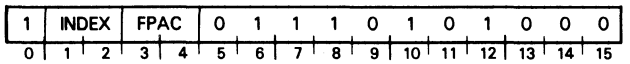
The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The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

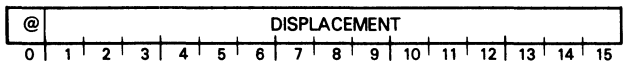
If the exponent processing produces either overflow or underflow, the corresponding bit in the floating point status register is set. The number in FPAC is correct except that, for exponent overflow, the exponent is 128 too small, and for exponent underflow, the exponent is 128 too large.

Divide Single (FPAC by Memory)

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

Valid for: CPU





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.

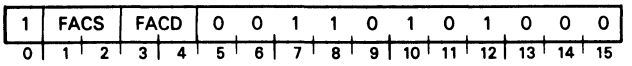
The mantissa in FPAC is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FPAC and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The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

Valid for: CPU



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.

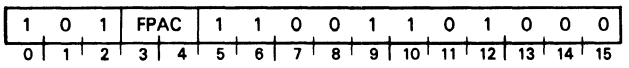
The mantissa in FACD is then divided by the mantissa of the source operand and the quotient is the mantissa of the intermediate result. The exponent of the source operand is subtracted from the exponent in FACD and 64 is added to this result. This addition of 64 maintains the *excess 64* notation. The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

Valid for: CPU



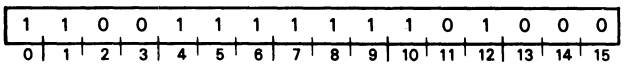
Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC. Ignores bits 0 and 8-15 of AC0. Leaves unchanged bits 0 and 8-63 of FPAC and the entire contents of AC0. Also sets bits 0-7 (the sign and exponent) to zero if bits 8-63 (the mantissa) of FPAC are zero. Leaves bits 1-7 of FPAC unchanged if FPAC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

Valid for: CPU



Raises the value, *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.

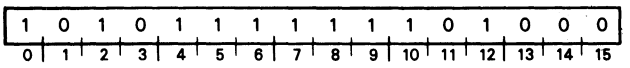
Format: Algorithm coefficients must follow the *Real Exponential* instructions . The standard format is:

RELATIVE 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	Error Return			

Real Exponential Single

FEXPS

Valid for: CPU



Raises the value, *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.

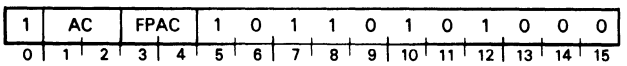
Format: Algorithm coefficients must follow the *Real Exponential* instructions . The standard format is:

RELATIVE 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	Error	Return

Fix To AC

FFAS *ac,fpac*

Valid for: CPU



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.

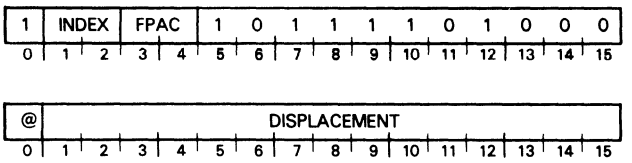
If the number in FPAC is less than -32,767 or greater than +32,767, this instruction sets the MOF flag in the floating point status register to 1.

> 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]*

Valid for: CPU



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.

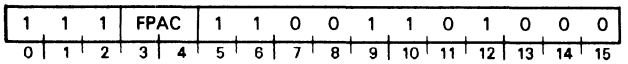
If the number in FPAC is less than -2,147,483,647 or greater than +2,147,483,647, this instruction sets the MOF flag in the floating point status register to 1.

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*

Valid for: CPU



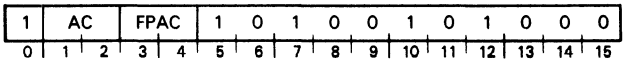
Divides the floating point number in FPAC by 2.

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*

Valid for: CPU



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

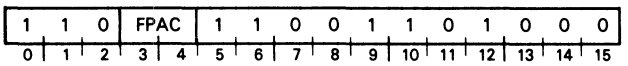
Converts the signed two's complement number contained in the specified accumulator to a single precision floating point number, places the result in the specified FPAC, and sets the low-order 32 bits of the FPAC to 0. Leaves the contents of the specified accumulator unchanged and destroys the previous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

Valid for: CPU



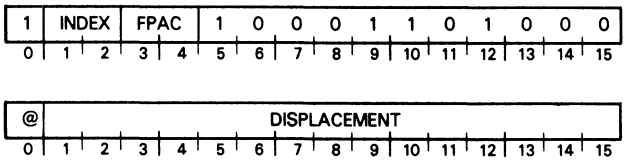
Zeros the fractional portion (if any) of the number contained in the specified FPAC and then normalizes the number. The instruction updates the Z and N flags in the floating point status register to reflect the new contents of the specified FPAC.

> 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]*

Valid for: CPU



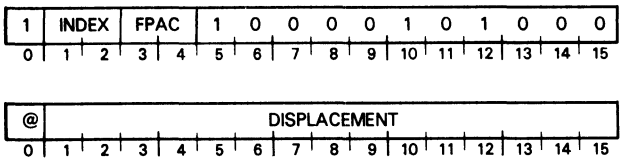
Moves four words out of memory into a specified FPAC.

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]*

Valid for: CPU



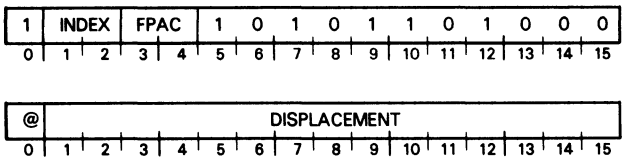
Moves two words out of memory into a specified FPAC.

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]*

Valid for: CPU



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

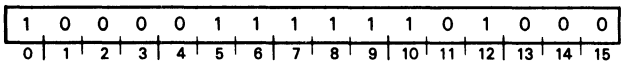
Computes the effective address *E*, converts the 32-bit, signed, two's complement number addressed by E to a double precision floating point number, and places the result in the specified FPAC. Destroys the previous contents of FPAC, and updates the Z and N flags in the floating point status register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

Valid for: CPU



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.

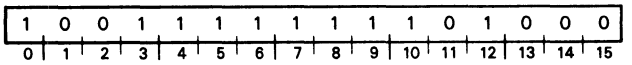
Format: Algorithm coefficients must follow the *Natural Logarithm* instructions. The format of the instruction is:

RELATIVE 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	Error Return (FPAC0 <= 0)			

Natural Logarithm Single

FLOGS

Valid for: CPU



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.

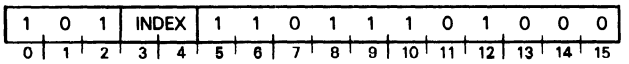
Format: Algorithm coefficients must follow the *Natural Logarithm* instructions. The format of the instruction is:

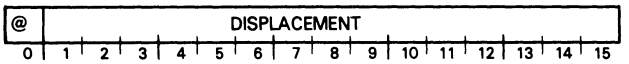
RELATIVE WORD	NAME	CODED VALUE (Hex)	
0	Instruction Word	FLOGS	
1-2	SQ.5	40B5	04F3
3-4	A1	40E5	4226
5-6	A0	C135	0453
7-8	B0	C11A	822A
9-10	LOG2	40B1	7218
11	Error	Error Return (FPAC0 <= 0)	

Load Floating Point Status

FLST *,[@]displacement[,index]*

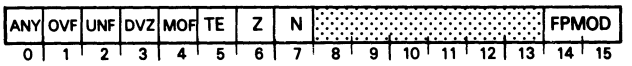
Valid for: CPU

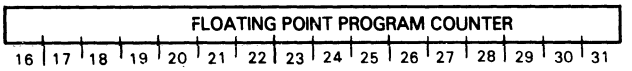




Moves the contents of two specified memory locations to the floating point status register.

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.



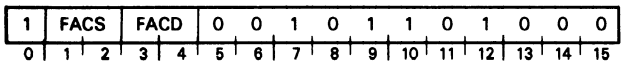


BITS	NAME	CONTENTS or FUNCTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator--while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator - while processing a floating point number, 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.
8-13	---	Reserved for future use.
14-15	FPMOD	Indicates computer series supporting the floating point instruction set. 00 S/200, C/300, S/230, C/330 01 S/130 Series 10 M/600 Series 11 Reserved for 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.

Multiply Double (FPAC by FPAC)

FMD *facs,facd*

Valid for: CPU



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 are set to reflect the new contents of FACD.

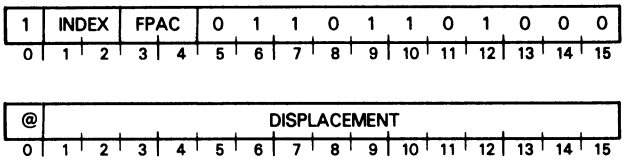
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 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]*

Valid for: CPU



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 are set to reflect the new contents of FPAC.

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

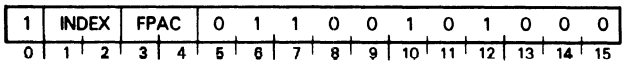
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 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.

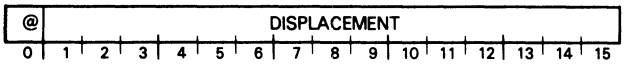
If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the corresponding flag in the floating point status register is set to 1. The number is correct except 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]*

Valid for: CPU





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 are set to reflect the new contents of FPAC.

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

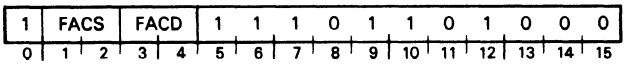
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 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*

Valid for: CPU



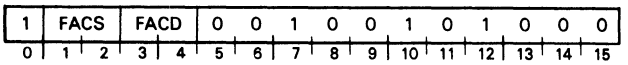
Moves the contents of one FPAC to another FPAC.

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*

Valid for: CPU



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 are set to reflect the new contents of FACD.

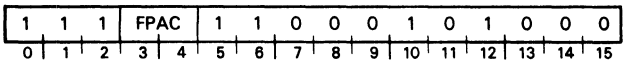
The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The exponents of the two numbers are added together and 64 is subtracted. This subtraction of 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*

Valid for: CPU

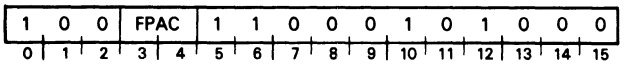


Inverts the sign bit of FPAC. Bits 1-63 of FPAC remain unchanged. Also sets the sign and exponent to zero if the mantissa in FPAC is zero. Updates the Z and N flags in the floating point status register to reflect the new contents of FPAC. If FPAC contains true zero, the sign bit remains unchanged.

Normalize

FNOM *fpac*

Valid for: CPU



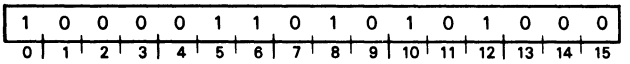
Normalizes the floating point numbers in FPAC. Sets a true zero in FPAC if all the bits of the mantissa are zero. Sets the UNF flag in the FPSR if an exponent underflow occurs. The number in FPAC is then correct, except that the exponent is 128 too large.

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

No Skip

FNS

Valid for: CPU

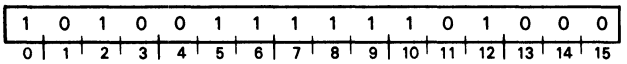


The next sequential word is executed.

Polynomial Evaluation Double

FPLYD

Valid for: CPU



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:

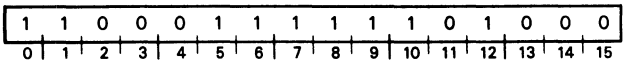
CODED WORD	MEANING
FPLYD	Instruction word
N	Nth degree, unnormalized
A_n	Coefficients
A_{n-1}	.
.	.
.	.
.	.
A_0	.
FPLYD	Instruction word
N + 100000_8	Nth degree, normalized
A_{n-1}	Coefficients
A_{n-2}	.
.	.
.	.
.	.
A_0	.

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

Valid for: CPU



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:

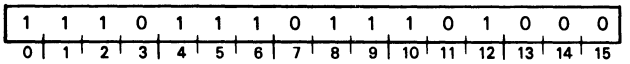
CODED WORD	MEANING
FPLYS	Instruction word
N	Nth degree, unnormalized
A_n	Coefficients
A_{n-1}	.
.	.
.	.
.	.
A_0	.
FPLYS	Instruction word
N + 100000_8	Nth degree, normalized
A_{n-1}	Coefficients
A_{n-2}	.
.	.
.	.
.	.
A_0	.

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.*

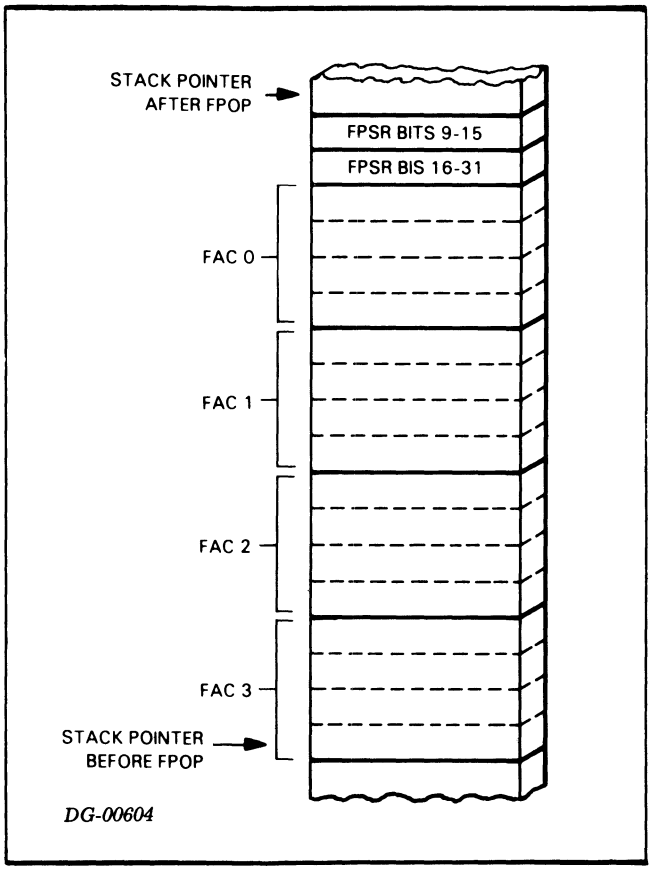
Pop Floating Point State

FPOP

Valid for: CPU



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:

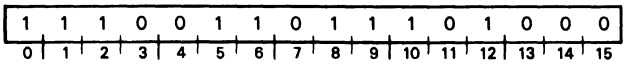


NOTE: *Because of the potentially long time required to perform some floating point instructions in relation to I/O interrupt requests, these instructions are interruptable. Because the FACD, stack pointer, and program counter are not updated until the completion of these instructions, any interrupt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

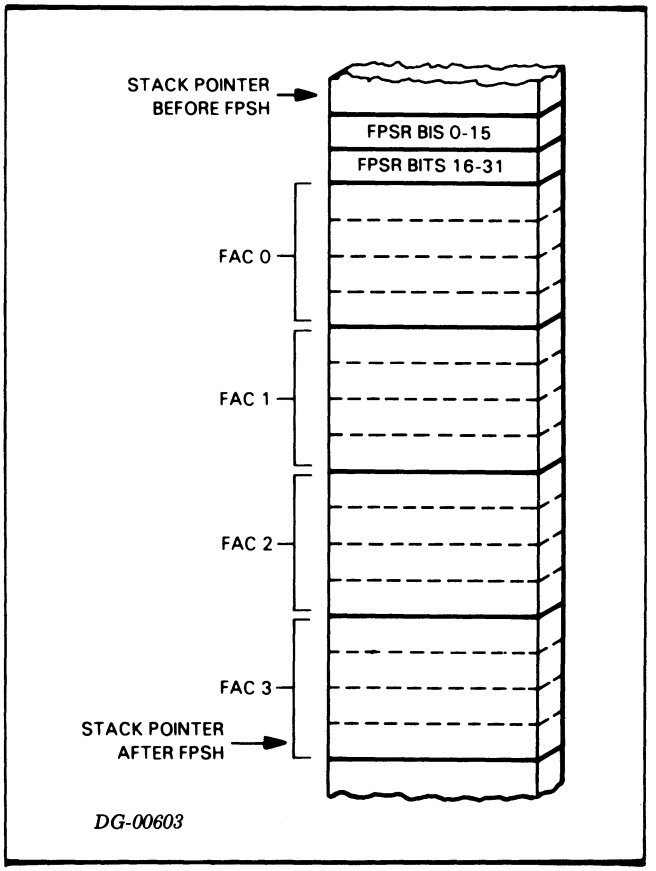
Push Floating Point State

FPSH

Valid for: CPU



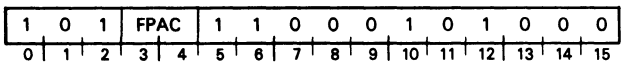
Pushes an 18-word floating point return block onto the user stack, leaving the contents of the floating point accumulators and the floating point status register unchanged. The format of the 18 words pushed is as follows:



Read High Word

FRH *fpac*

Valid for: CPU

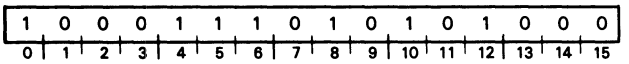


Places the high-order 16 bits of FPAC in AC0, destroys the previous contents of AC0, and leaves unchanged the contents of FPAC and the Z and N flags in the floating point status register.

Skip Always

FSA

Valid for: CPU

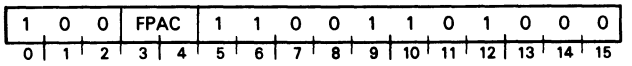


The next sequential word is skipped.

Scale

FSCAL *fpac*

Valid for: CPU



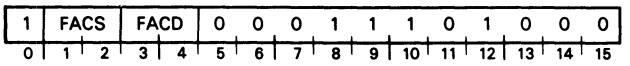
Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0. Leaves the contents of AC0 unchanged.

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.

Subtract Double (FPAC from FPAC)

FSD *facs,facd*

Valid for: CPU



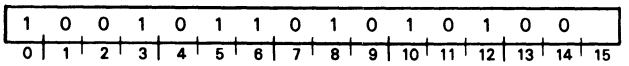
Subtracts the floating point number in FACS from the floating point number in FACD and places the normalized result in the FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents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

Skip On Zero

FSEQ

Valid for: CPU

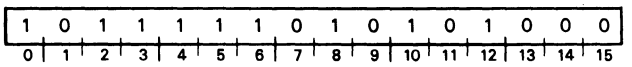


Skips the next sequential word if the Z flag of the floating point status register is 1.

Skip On Greater Than Zero

FSGT

Valid for: CPU

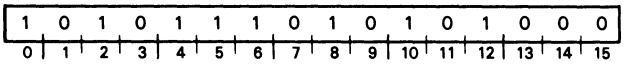


Skips the next sequential word if both the Z and N flags of the floating point status register are 0.

Skip On Greater Than Or Equal To Zero

FSGE

Valid for: CPU

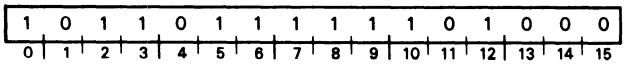


Skips the next sequential word if the N flag of the floating point status register is 0.

Sine Double

FSIND

Valid for: CPU



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.

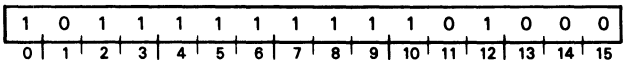
Format: Algorithm coefficients must follow the *Sine* instruction. The format is:

RELATIVE 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

Valid for: CPU



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.

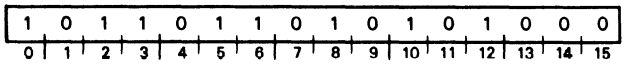
Format: Algorithm coefficients must follow the *Sine* instruction. The format is:

RELATIVE 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

Valid for: CPU

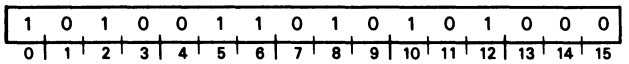


Skips the next sequential instruction if either the Z flag or the N flag of the floating point status register is 1.

Skip On Less Than Zero

FSLT

Valid for: CPU

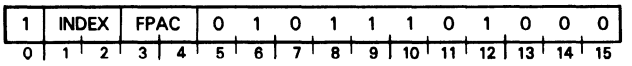


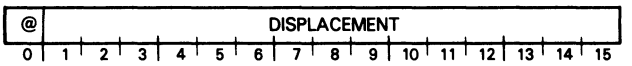
Skips the next sequential word if the N flag of the floating point status register is 1.

Subtract Double (Memory from FPAC)

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

Valid for: CPU





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.

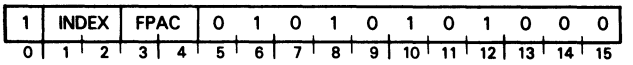
The instruction computes the effective address *E* which addresses a 4-word (double precision) operand.

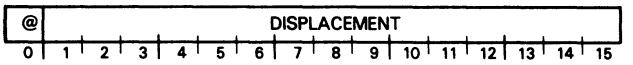
The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition. (*See* FAMD.)

Subtract Single (Memory from FPAC)

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

Valid for: CPU





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.

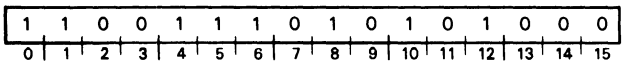
The instruction computes the effective address *E* which addresses a 2-word (single precision) operand.

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

Valid for: CPU

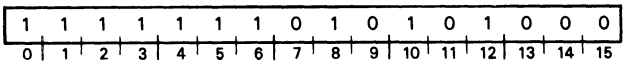


Skips the next sequential word if the divide by zero (DVZ) flag of the floating point status register is 0.

Skip On No Error

FSNER

Valid for: CPU

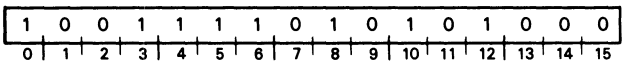


Skips the next sequential word if bits 1-4 of the floating point status register are all 0.

Skip On Non-Zero

FSNE

Valid for: CPU

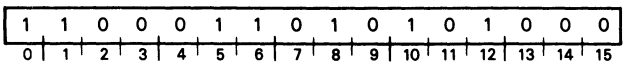


Skips the next sequential word if the Z flag of the floating point status register is 0.

Skip On No Mantissa Overflow

FSNM

Valid for: CPU

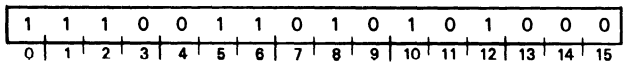


Skips the next sequential word if the mantissa overflow (MOF) flag of the floating point status register is 0.

Skip On No Overflow

FSNO

Valid for: CPU

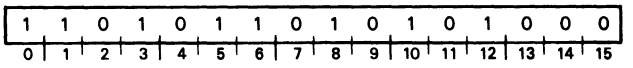


Skips the next sequential word if the overflow (OVF) flag of the floating point status register is 0.

Skip On No Underflow

FSNU

Valid for: CPU

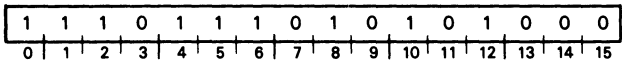


Skips the next sequential word if the underflow (UNF) flag of the floating point status register is 0.

Skip On No Overflow and No Zero Divide

FSNOD

Valid for: CPU

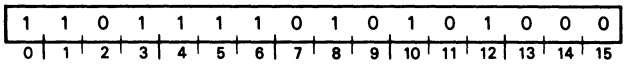


Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the floating point status register are 0.

Skip On No Underflow And No Zero Divide

FSNUD

Valid for: CPU

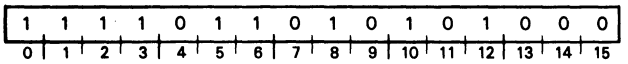


Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the floating point status register are 0.

Skip On No Underflow And No Overflow

FSNUO

Valid for: CPU

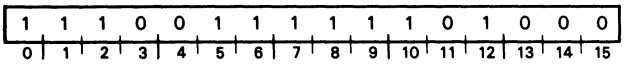


Skips the next sequential word if both the underflow (UNF) flag and overflow (OVF) flag of the floating point status register are 0.

Square Root Double

FSQRD

Valid for: CPU



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.

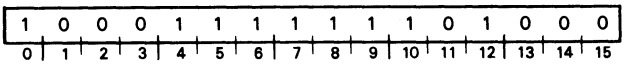
Format: Use the following format:

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

Square Root Single

FSQRS

Valid for: CPU



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.

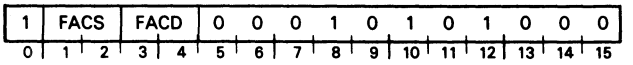
Format: Use the following format:

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

Subtract Single (FPAC from FPAC)

FSS *facs,facd*

Valid for: CPU



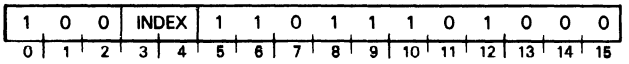
Subtracts the floating point number in FACS from the floating point number in FACD and places the normalized result in the FACD. Destroys the previous contents of FACD, leaves the contents of FACS unchanged, and updates the Z and N flags in the floating point status register to reflect the new contents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

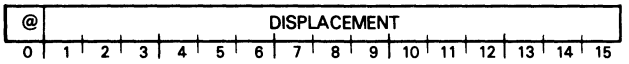
The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating point addition.

Store Floating Point Status

FSST *,[@]displacement[,index]*

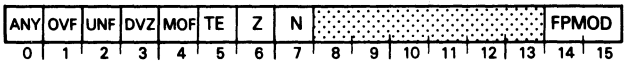
Valid for: CPU

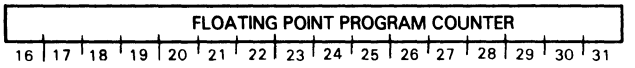




Moves the contents of the FPSR to two specified memory locations.

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.



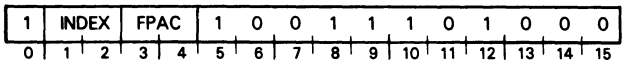


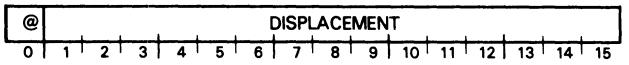
BITS	NAME	CONTENTS or FUNCTION
0	ANY	Indicates that any of bits 1-4 are set.
1	OVF	Overflow Indicator--while processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow Indicator - while processing a floating point number, 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.
8-13	---	Reserved for future use.
14-15	FPMOD	Indicates computer series supporting the floating point instruction set. 00 S/200, C/300, S/230, C/330 01 S/130 Series 10 M/600 Series 11 Reserved for 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.

Store Floating Point Double

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

Valid for: CPU





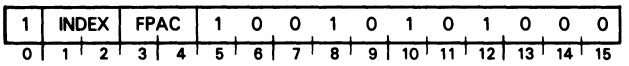
Stores the contents of a specified FPAC into a memory location.

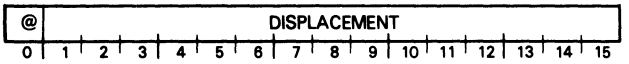
Computes the effective address, *E*, and places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location and leaves unchanged the contents of FPAC and the condition codes in the FPSR.

Store Floating Point Single

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

Valid for: CPU





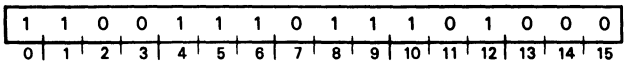
Stores the contents of a specified FPAC into a memory location.

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 addresed 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.

Trap Disable

FTD

Valid for: CPU



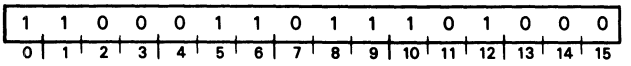
Sets the trap enable bit of the FPSR to 0.

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

Trap Enable

FTE

Valid for: CPU



Sets the trap enable bit of the FPSR to 1.

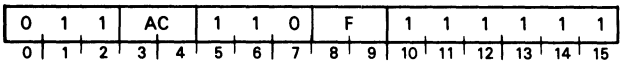
NOTE: *When a floating point fault occurs and the trap enable bit is 1, the trap enable bit is set to 0 before control is transferred to the floating point error handler. The trap enable bit should be set to 1 before normal processing is resumed.*

Halt

HALTA *ac*

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

Valid for: CPU - IOP - DCU



Stops the processor.

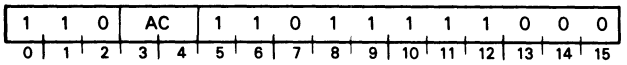
Sets the Interrupt On flag according to the function specified by F, then stops the processor. The data lights display the contents of the specified accumulator.

NOTE: *The assembler recognizes the mnemonic* **HALT** *as equivalent to the instruction* **HALTA 0**.

Halve

HLV *ac*

Valid for: CPU - IOP



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

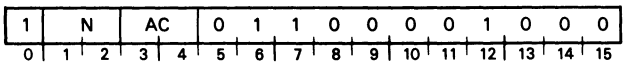
The signed, 16-bit two's complement number contained in the specified AC is divided by 2 and rounded toward 0. The result is placed in the specified AC.

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 Left

HXL *n,ac*

Valid for: CPU - IOP



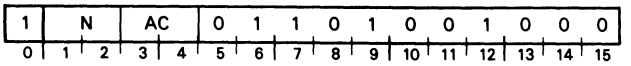
Shifts the contents of AC left a number of hex digits depending upon the immediate field **N**. The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of AC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

Hex Shift Right

HXR *n,ac*

Valid for: CPU - IOP



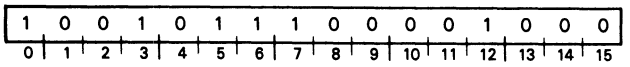
Shifts the contents of AC right a number of hex digits depending upon the immediate field, **N**. The number of digits shifted is equal to N+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If N is equal to 3, then all 16 bits of AC are shifted out and all bits of AC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

Load Map (IOP)

LMP

Valid for: - IOP



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 IOP map is shown below.

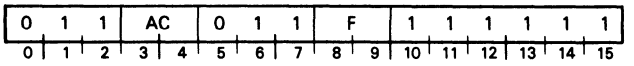
BITS	NAME	FUNCTION
0	HOST	0 = map page into local memory. 1 = map page into host memory.
1-5	PAGE	Logical page number.
6-14	---	Reserved for future use.
15	DCH LD	0 = load entry into USER map. 1 = load entry into DCH map.

Interrupt Acknowledge

INTA

DIB*[f]* *ac*,CPU

Valid for: CPU - IOP - DCU



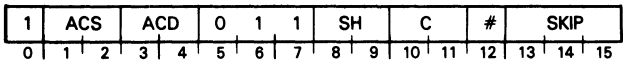
Returns device code of an interrupting device.

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.

Increment

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

Valid for: CPU - IOP - DCU



Increments the contents of an accumulator.

Initializes the carry bit to the specified value. Increments the unsigned, 16-bit number in ACS by one and places the result in the shifter. If the incrementation produces a carry of 1 out of the high order bit, the instruction complements the carry bit. Performs the specified shift operation, and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

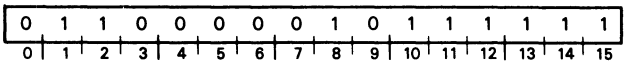
NOTE: *If the number in ACS is 177777_8 the instruction complements the carry bit.*

Interrupt Disable

INTDS

NIOC CPU

Valid for: CPU - IOP - DCU



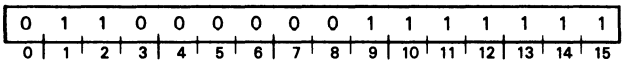
Sets Interrupt On flag to 0.

Interrupt Enable

INTEN

NIOS CPU

Valid for: CPU - IOP - DCU



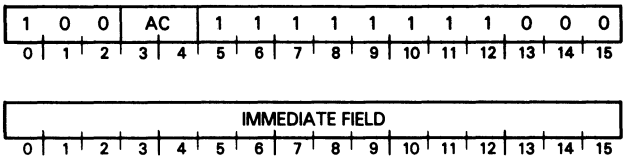
Sets Interrupt On flag to 1.

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 interruptable, then interrupts can occur as soon as the instruction begins to execute.

Inclusive OR Immediate

IORI *i,ac*

Valid for: CPU - IOP

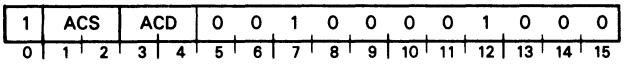


Forms the logical inclusive OR of the contents of the immediate field and the contents of the specified AC and places the result in the specified AC.

Inclusive OR

IOR *acs,acd*

Valid for: CPU - IOP



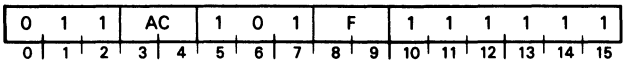
Forms the logical inclusive OR of the contents of ACS and the contents of ACD and places the result in ACD. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the instruction sets the result bit to 0. The contents of ACS remain unchanged.

Reset

IORST

DIC*[f]* *ac*,CPU

Valid for: CPU - IOP - DCU



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

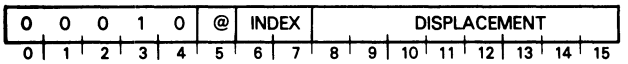
Sets the Busy and Done flags in all I/O devices to 0. Sets the 16-bit priority mask to 0. Sets the Interrupt On flag according to the function specified by F.

> **NOTE:** *The assembler recognizes the mnemonic* **IORST** *as equivalent to the instruction* **DICC 0,CPU.**
>
> *If the mnemonic* **DIC** *is used to perform this function, you must code an accumulator to avoid assembly errors. During execution, the accumulator field is ignored and the contents of the accumulator remain unchanged.*

Increment And Skip If Zero

ISZ *,[@]displacement[,index]*

Valid for: CPU - IOP - DCU



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 places the address of the next sequential instruction in the program counter and operation continues from there.

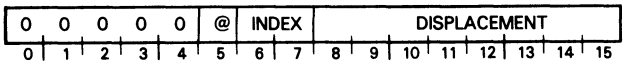
NOTE: *Issued to the CPU, this instruction both reads and possibly modifies the contents of a memory location in a single memory operation. Issued by an IOP or a DCU, the instruction may read and modify memory in separate memory operations if the effective address* E *lies outside that processor's local physical memory space. Each such separate memory operation propagates to host memory via the host data channel. All logical addresses above* 1777_8 *in the DCU, and memory references mapped to the host from the IOP lie outside local physical memory.*

Contention on the data channel, or activity on the burst multiplexor channel may interleave the two data channel transfers. It is possible for the interleaving operation to alter one memory location in the time between the read and the modify cycles of the remotely-issued instruction.

Jump

JMP

Valid for: CPU - IOP - DCU

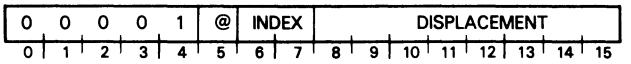


Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Jump To Subroutine

JSR *,[@]displacement[,index]*

Valid for: CPU - IOP - DCU



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

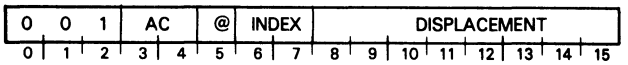
Computes the effective address, *E*; then places the address of the next sequential instruction in AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

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

Load Accumulator

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

Valid for: CPU - IOP - DCU



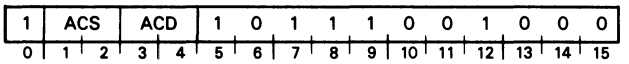
Copies a word from memory to an accumulator.

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*

Valid for: CPU - IOP



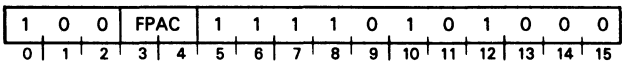
Moves a byte from memory (as addressed by a byte pointer in one accumulator) to the second accumulator.

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

Load Integer

LDI *fpac*

Valid for: CPU



Translates a decimal integer from memory to (normalized) floating point format and places the result in a floating point accumulator.

Under the control of accumulators AC1 and AC3, converts a decimal integer to floating point form, normalizes it, and places it in the specified FPAC. The instruction updates the Z and N bits in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory, and destroys the previous contents of the specified FPAC.

AC1 must contain the data-type indicator describing the number.

AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Numbers of data type 7 are not normalized after loading. By convention, the first byte of a number stored according to data type 7 must contain the sign and exponent of the floating point number. The exponent must be in "excess 64" representation. The instruction copies each byte (following the lead byte) directly to mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data from memory.

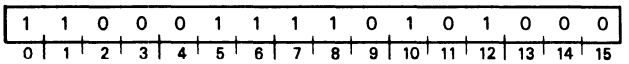
Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3; the contents of AC3 are undefined.

NOTE: *An attempt to load a minus 0 sets the specified FPAC to true zero.*

Load Integer Extended

LDIX

Valid for: CPU



Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into 4 units of 8 digits each and converts each unit to a floating point number. Places the number obtained from the 8 high-order digits into FAC0, the number obtained from the next 8 digits into FAC1, the number obtained from the next 8 digits into FAC2, and the number obtained from the low-order 8 bits into FAC3. The instruction places the sign of the integer in each FPAC unless that FPAC has received 8 digits of zeros, in which case the instruction sets FPAC to true zero. The Z and N flags in the floating point status register are unpredictable.

AC1 must contain the data-type indicator describing the integer.

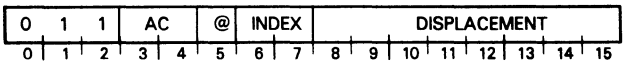
AC3 must contain a byte pointer which is the address of the high-order byte of the integer.

Upon successful termination, the contents of AC0 and AC3 are undefined; the contents of AC1 remain unchanged; and AC2 contains the original contents of AC3.

Load Effective Address

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

Valid for: CPU - IOP



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 an auto-incrementing or auto-decrementing location is referenced in the course of the effective address calculation, the contents of the location are incremented or decremented. Note, however, that auto-incrementing and auto-decrementing is suppressed when demand paging is enabled.

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.

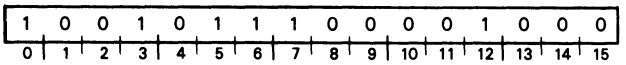
```
LEF     0,TABLE     ; The logical address of
                    ; TABLE is placed in AC0.
LEF     1,-55,3     ; Subtracts 000055 (octal)
                    ; from the unsigned integer
                    ; in AC3 and the result is
                    ; placed in AC1.
LEF     0,. +0      ; Places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

NOTE: *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.*

Load Map

LMP

Valid for: CPU



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.

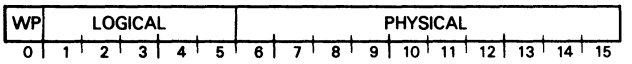
AC1 must contain an unsigned integer which is the number of words to be loaded into the MAP. Bits 1-15 of AC2 must contain the address of the first word to be loaded. If bit 0 of AC2 is 1, the instruction follows the indirection chain and places the resultant effective address in AC2. AC0 and AC3 are 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 interruptable 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:



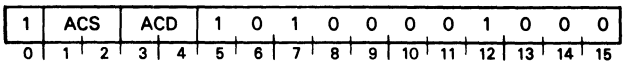
BITS	NAME	CONTENTS or FUNCTION
0	WP	Unused for data channel maps; write protect for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

NOTE: *Declare a logical page invalid by setting the Write Protect bit to 1 and all of bits 6-15 to 1.*

Locate Lead Bit

LOB *acs,acd*

Valid for: CPU - IOP



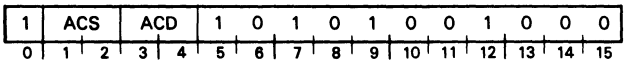
Adds a number equal to the number of high-order zeroes in the contents of ACS to the signed, 16-bit, two's complement number contained in ACD. The contents of ACS and the state of the carry bit remain unchanged.

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*

Valid for: CPU - IOP



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 the carry bit 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.*

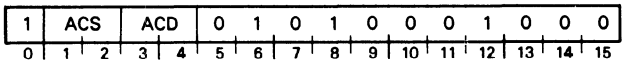
NOTE: *Issued to the CPU, this instruction both reads and possibly modifies the contents of a memory location in a single memory operation. Issued by an IOP or a DCU, the instruction may read and modify memory in separate memory operations if the effective address* E *lies outside that processor's local physical memory space. Each such separate memory operation propagates to host memory via the host data channel. All logical addresses above* 1777_8 *in the DCU, and memory references mapped to the host from the IOP lie outside local physical memory.*

Contention on the data channel, or activity on the burst multiplexor channel may interleave the two data channel transfers. It is possible for the interleaving operation to alter one memory location in the time between the read and the modify cycles of the remotely-issued instruction.

Logical Shift

LSH *acs,acd*

Valid for: CPU - IOP



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.

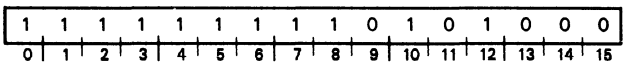
The number of bits shifted is equal to the magnitude of the number in bits 8-15 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. The carry bit and the contents of ACS remain unchanged.

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. The carry bit and the contents of ACS remain unchanged.*

Load Sign

LSN

Valid for: CPU



Under the control of accumulators AC1 and AC3, evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

VALUE OF NUMBER	CODE
Positive non-zero	+1
Negative non-zero	-1
Positive zero	0
Negative zero	-2

AC1 must contain the data type indicator describing the number.

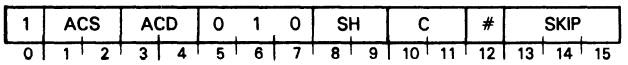
AC3 must contain a byte pointer which is the address of the high-order byte of the number.

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. The contents of the addressed memory locations remain unchanged.

Move

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

Valid for: CPU - IOP - DCU



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

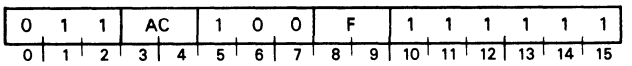
Initializes the carry bit to the specified value. Places the contents of ACS in the shifter. Performs the specified shift operation and loads the result of the shift into ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

Mask Out

MSKO

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

Valid for: CPU - IOP - DCU



Sets the priority mask.

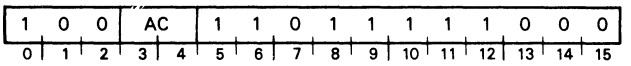
Places the contents of the specified accumulator in the priority mask. After the transfer, sets the Interrupt On flag according to the function specified by F. The contents of the specified AC remain unchanged.

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

Modify Stack Pointer

MSP *ac*

Valid for: CPU - IOP



Changes the value of the stack pointer and tests for potential overflow.

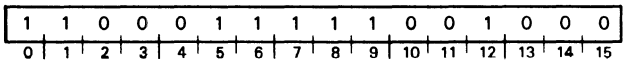
Adds the signed two's-complement number in AC to the stack pointer. If the result is less than the stack limit, the instruction places the result in the stack pointer.

If the result is greater than the stack limit, the instruction transfers control to the stack fault routine. The program counter in the fault return block is the address of the *Modify Stack Pointer* instruction. The stack pointer is left unchanged.

Unsigned Multiply

MUL

Valid for: CPU - IOP



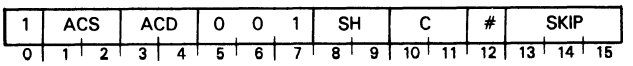
Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

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.

Negate

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

Valid for: CPU - IOP - DCU



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

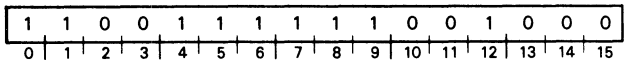
Initializes the carry bit to the specified value. Places the two's complement of the unsigned, 16-bit number in ACS in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the instruction complements the carry bit. Performs the specified shift operation and places the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

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

Signed Multiply

MULS

Valid for: CPU - IOP



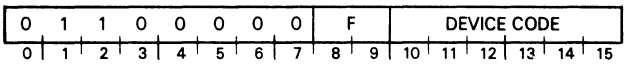
Multiplies the signed contents of two accumulators and adds the result to the signed contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

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.

No I/O Transfer

NIO *[f] device*

Valid for: CPU - IOP - DCU



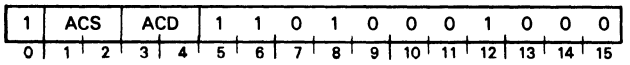
Used when a Busy or Done flag must be changed with no other operation taking place.

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

Pop Multiple Accumulators

POP *acs,acd*

Valid for: CPU - IOP



Pops 1 to 4 words off the stack and places them in the indicated accumulators.

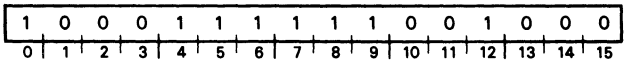
The set of accumulators from ACS through ACD is filled with words popped from the stack. The accumulators are filled in descending order, starting with the AC specified by ACS and continuing down through the AC specified by ACD, wrapping around if necessary, with AC3 following AC0. If ACS is equal to 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

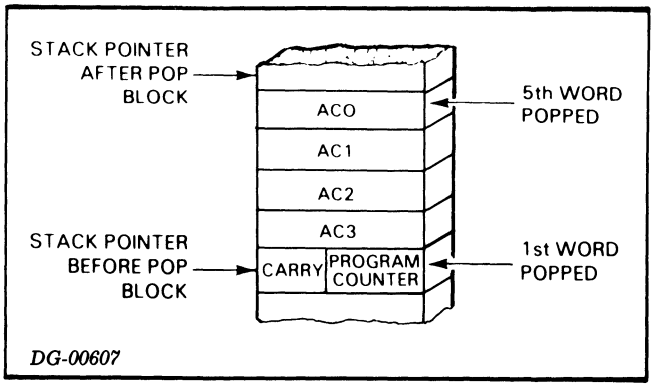
POPB

Valid for: CPU - IOP



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:



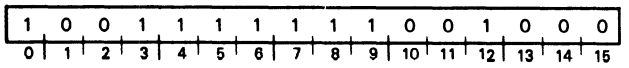
Sequential operation is continued with the word addressed by the updated value of the program counter.

> **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

Valid for: CPU - IOP

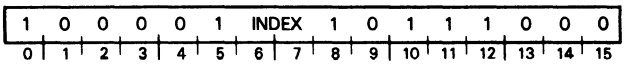


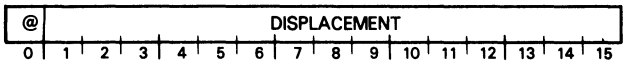
Pops the top word off the stack and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Push Jump

PSHJ *,[@]displacement[,index]*

Valid for: CPU - IOP



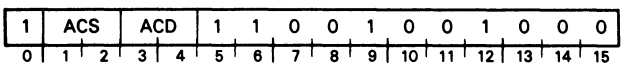


Pushes the address of the next sequential instruction onto the stack, computes the effective address *E* and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

Push Multiple Accumulators

PSH *acs,acd*

Valid for: CPU - IOP



Pushes the contents of 1 to 4 accumulators onto the stack.

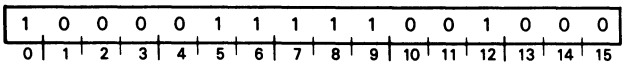
The set of accumulators from ACS through ACD is pushed onto the stack. The accumulators are pushed in ascending order, starting with the AC specified by ACS and continuing up through the AC specified by ACD, wrapping around if necessary, with AC0 following AC3. The contents of the accumulators remain 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 Return Address

PSHR

Valid for: CPU - IOP



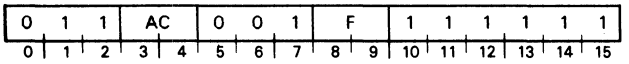
Pushes the address of this instruction *plus 2* onto the stack.

Read Switches

READS *ac*

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

Valid for: CPU - IOP



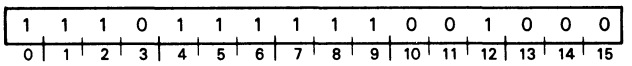
Places the contents of the console switches into an accumulator.

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.

Restore

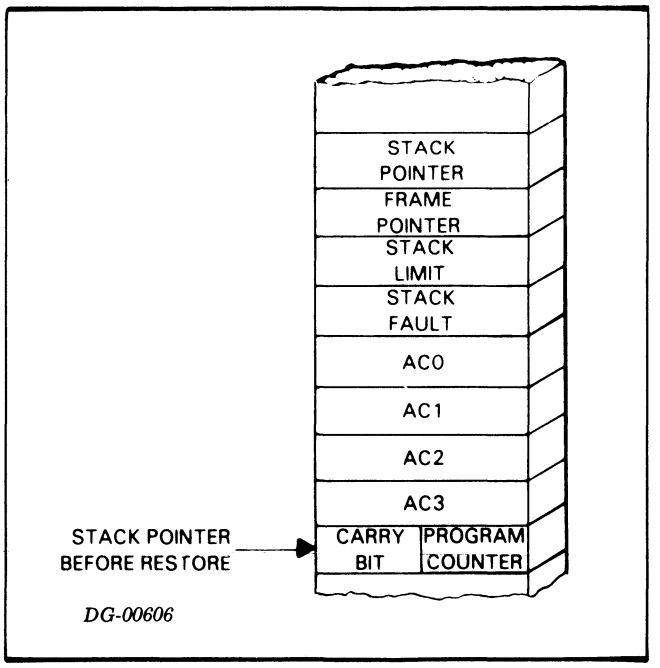
RSTR

Valid for: CPU - IOP



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.

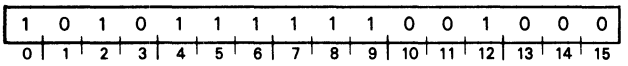
NOTE: *Use the* Restore *instruction to return control to the program only if the I/O interrupt handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction.*

The Restore *instruction does not check for stack underflow.*

Return

RTN

Valid for: CPU - IOP



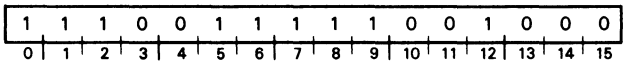
Returns control from subroutines that issue a *Save* instruction at their entry points.

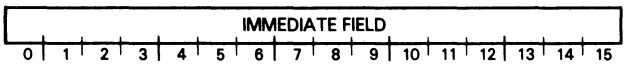
The contents of the frame pointer are placed in the stack pointer and a *Pop Block* instruction is executed. The popped value of AC3 is placed in the frame counter.

Save

SAVE *i*

Valid for: CPU - IOP





Saves the information required by the RETURN instruction.

A return block is pushed onto the stack. After the fifth word of the return block is pushed, the value of the stack pointer is placed in the frame pointer and in AC3. The 16-bit unsigned integer (called the *frame size*) contained in the immediate field is added to the stack pointer. The format of the five words pushed is as follows:

WORD PUSHED	CONTENTS
1	AC0
2	AC1
3	AC2
4	Frame pointer before the save
5	Bit 0 = carry bit Bits 1-15 = bits 1-15 of AC3

The *Save* instruction allocates a portion of the stack for use by the procedure which executed the *Save* . The value of the *frame size* 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.

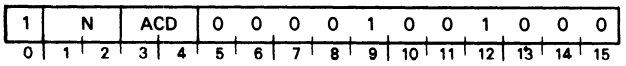
Before execution, the *Save* instruction checks for stack overflow. If executing the instruction would result in a stack overflow, *Save* transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save* instruction.

Use the *Save* instruction with the *Jump to Subroutine* instruction, which places the return value of the program counter in AC3. *Save* then pushes the return value (contents of AC3) into bits 1-15 of the fifth word pushed.

Subtract Immediate

SBI *n,ac*

Valid for: CPU - IOP

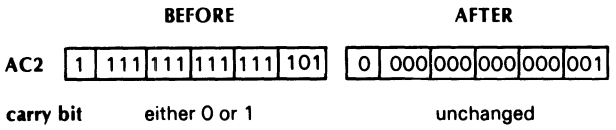


Subtracts an unsigned integer in the range 1-4 from the contents of an accumulator.

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. The carry bit 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.*

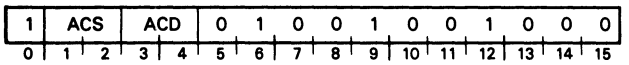
Example - Assume that AC2 contains 000003_8. After the instruction **SBI 4,2** is executed, AC2 contains 177777_8 and carry bit remains unchanged.



Skip If ACS Greater Than Or Equal to ACD

SGE *acs,acd*

Valid for: CPU - IOP



Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

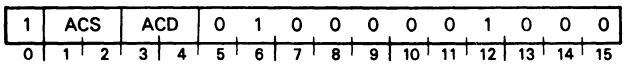
The signed two's complement numbers in ACS and ACD are algebraically compared. If the number in ACS is greater than or equal to the number in ACD, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

> 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*

Valid for: CPU - IOP



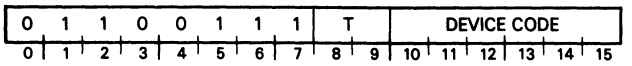
Compares two signed integers in two accumulators and skips if the first is greater than the second.

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.

I/O Skip

SKP *[t] device*

Valid for: CPU - IOP - DCU

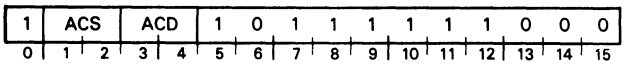


If the test condition specified by T is true, the instruction skips the next sequential word.

Skip On Non-Zero Bit

SNB *acs,acd*

Valid for: CPU - IOP



The two accumulators form a bit pointer. If the addressed bit is 1, the next sequential word is skipped.

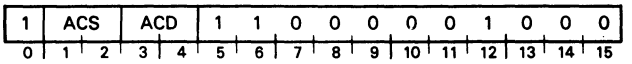
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer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

Store Byte

STB *acs,acd*

Valid for: CPU - IOP



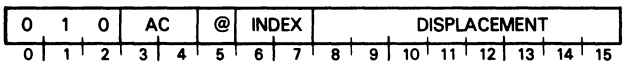
Moves the right byte of one accumulator to a byte in memory. The second accumulator contains the byte pointer.

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

Store Accumulator

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

Valid for: CPU - IOP - DCU



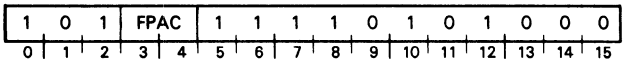
Stores the contents of an accumulator into a memory location.

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.

Store Integer

STI *fpac*

Valid for: CPU



Under the control of accumulators AC1 and AC3, translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory beginning at the specified location. The instruction leaves the floating point number unchanged in the FPAC, and destroys the previous contents of memory at the specified location(s).

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3 and AC3 contains a byte pointer which is the address of the next byte after the destination field.

NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction to clear any fractional part.*

If the destination field cannot contain the entire number being stored, high-order digits are discarded until the number will fit into the destination. The remaining low-order digits are stored and Carry is set to 1.

For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the high-order bytes to the right of the sign are set to 0.

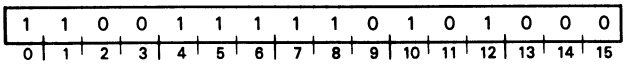
For data type 6, if the number being stored will not fill the destination field, the sign bit is extended to the left to fill the field.

For data type 7, if the number being stored will not fill the destination field, the low-order bytes are set to 0.

Store Integer Extended

STIX

Valid for: CPU



Converts the contents of the four FPAC's to integer form and uses the low-order 8 digits of each to form a 32-digit integer. The instruction stores this integer, right-justified, in memory beginning at the specified location. The sign of the integer is the logical OR of the signs of all four FPAC's. The previous contents of the addressed memory locations are lost. Sets the carry bit to 0. The contents of the FPAC's remain unchanged. The condition codes in the FPSR are unpredictable.

AC1 must contain the data-type indicator describing the form of the in memory.

AC3 must contain a byte pointer which is the address of the high-order byte of the destination field in memory.

Upon successful termination, the contents of AC0 are undefined; the contents of AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer which is the address of the next byte after the destination field.

NOTES: *If the destination field is not large enough to contain the number being stored, the instruction disregards high-order digits until the number will fit in the destination. The instruction stores low-order digits remaining and sets the carry bit to 1.*

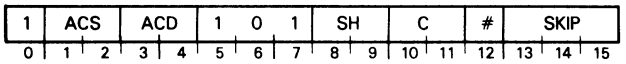
For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the instruction sets the high-order bytes to 0.

For data type 6, if the number being stored will not fill the destination field, the instruction extends the sign bit to the left to fill the field.

Subtract

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

Valid for: CPU - IOP - DCU



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

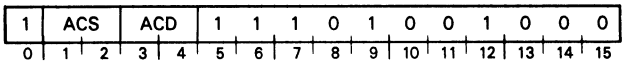
Initializes the carry bit to its specified value. The instruction subtracts the unsigned, 16-bit number in ACS from the unsigned, 16-bit number in ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. The instruction places the result of the addition in the shifter.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

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

System Call

SYC *acs,acd*

Valid for: CPU - IOP



Pushes a return block and indirectly places the address of the *system call handler* in the program counter.

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 executed a *jump indirect* to location 2. If this instruction disabled a user map, then I/O interrupts cannot occur between the time the *System call* instruction is executed and the time the instruction pointed to by the contents of location 2 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. If either of the accumulators specified is not AC0, then the instruction takes no special action. The contents of the specified accumulators remain unchanged.*

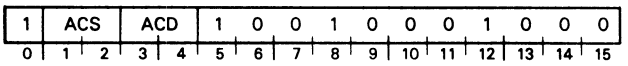
The assembler recognizes the mnemonic **SCL** *as equivalent to* **SYC 1,1**.

The assembler recognizes the mnemonic **SVC** *as equivalent to* **SYC 0,0**.

Skip On Zero Bit

SZB *acs,acd*

Valid for: CPU - IOP



The two accumulators form a bit pointer. If the addressed bit is zero, the next sequential word is skipped.

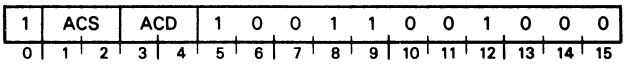
Forms a 32-bit bit pointer from the contents of ACS and ACD. ACS contains the high-order 16 bits and ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer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

Skip On Zero Bit And Set To One

SZBO *acs,acd*

Valid for: CPU - IOP



The two accumulators form a bit pointer. If the addressed bit is 0, the instruction skips the next sequential word. The instruction sets the addressed 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 sets the addressed bit in memory to 1. If the bit was 0 before being set to 1, the instruction skips the next sequential word. The contents of ACS and ACD remain unchanged.

NOTE: *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.*

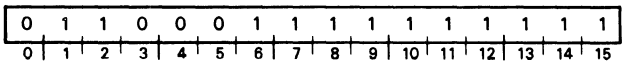
NOTE: *Issued to the CPU, this instruction both reads and possibly modifies the contents of a memory location in a single memory operation. Issued by an IOP or a DCU, the instruction may read and modify memory in separate memory operations if the effective address* E *lies outside that processor's local physical memory space. Each such separate memory operation propagates to host memory via the host data channel. All logical addresses above* 1777_8 *in the DCU, and memory references mapped to the host from the IOP lie outside local physical memory.*

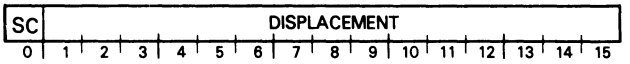
Contention on the data channel, or activity on the burst multiplexor channel may interleave the two data channel transfers. It is possible for the interleaving operation to alter one memory location in the time between the read and the modify cycles of the remotely-issued instruction.

Vector On Interrupting Device Code

VCT *[@]displacement[,index]*

Valid for: CPU - IOP





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 then used as a pointer to the appropriate interrupt handler (Mode A) or as a pointer to another table which points to the interrupt handler and contains a new priority mask (Modes B through E). The instruction can also save the state of the machine by pushing various words onto the stack, creating a new vector stack, and setting up a priority structure.

The accompanying flow chart (*see opposite page*) 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:

Direct Bit - Bit 0 of the selected vector table entry;

Stack Change Bit - Bit 0 of the second word of the *Vector* instruction;

Push Bit - Bit 0 of the first word of the selected device control table.

The state of these bits collectively determine which mode will be used by the *Vector* instruction. This relationship is 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 functions performed by the *Vector* instruction within each mode are summarized here:

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 the carry bit) 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 the process through step-by-step.

Common Process

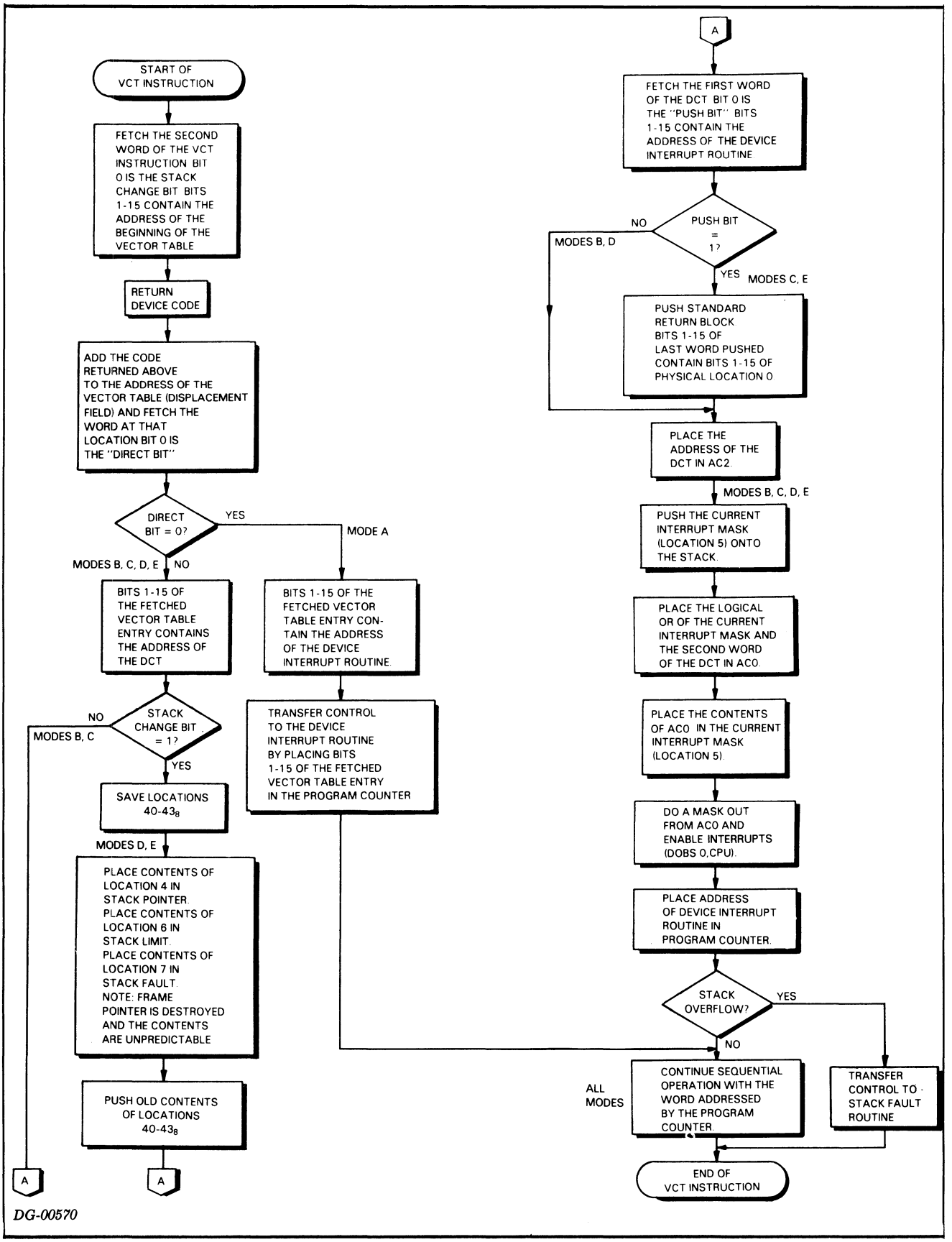
The initial steps taken by the *Vector* instruction are done regardless of the mode being used. The device code of the interrupting device is returned. This code is added to the address of the start of the vector table, which is found in the displacement field (bits 1-15 of the second instruction word), to get a new address within the vector table. The word at this new location is fetched and its bit 0 (the direct bit) is examined.

Mode A

If the direct bit is 0, mode A is used and the state of the other control bits does not matter. Bits 1-15 of the fetched vector table entry are used as the address of the interrupt handler for the interrupting device. Control is immediately transferred to the interrupt handler.

Mode B

Modes B through E perform different functions initially, but use a common second part. We discuss the common second part after discussing each Part I separately.


START OF VCT INSTRUCTION
FETCH THE SECOND WORD OF THE VCT INSTRUCTION. BIT 0 IS THE STACK CHANGE BIT. BITS 1-15 CONTAIN THE ADDRESS OF THE BEGINNING OF THE VECTOR TABLE
RETURN DEVICE CODE
ADD THE CODE RETURNED ABOVE TO THE ADDRESS OF THE VECTOR TABLE (DISPLACEMENT FIELD) AND FETCH THE WORD AT THAT LOCATION. BIT 0 IS THE "DIRECT BIT"
DIRECT BIT = 0?
YES
MODE A
MODES B, C, D, E
NO
BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY CONTAIN THE ADDRESS OF THE DCT
BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY CONTAIN THE ADDRESS OF THE DEVICE INTERRUPT ROUTINE
STACK CHANGE BIT = 1?
NO
MODES B, C
YES
TRANSFER CONTROL TO THE DEVICE INTERRUPT ROUTINE BY PLACING BITS 1-15 OF THE FETCHED VECTOR TABLE ENTRY IN THE PROGRAM COUNTER
SAVE LOCATIONS $40-43_8$
MODES D, E
PLACE CONTENTS OF LOCATION 4 IN STACK POINTER.
PLACE CONTENTS OF LOCATION 6 IN STACK LIMIT.
PLACE CONTENTS OF LOCATION 7 IN STACK FAULT.
NOTE: FRAME POINTER IS DESTROYED AND THE CONTENTS ARE UNPREDICTABLE
PUSH OLD CONTENTS OF LOCATIONS $40-43_8$
A
A
A
FETCH THE FIRST WORD OF THE DCT. BIT 0 IS THE "PUSH BIT". BITS 1-15 CONTAIN THE ADDRESS OF THE DEVICE INTERRUPT ROUTINE.
PUSH BIT = 1?
NO
MODES B, D
YES
MODES C, E
PUSH STANDARD RETURN BLOCK. BITS 1-15 OF LAST WORD PUSHED CONTAIN BITS 1-15 OF PHYSICAL LOCATION 0.
PLACE THE ADDRESS OF THE DCT IN AC2.
MODES B, C, D, E
PUSH THE CURRENT INTERRUPT MASK (LOCATION 5) ONTO THE STACK.
PLACE THE LOGICAL OR OF THE CURRENT INTERRUPT MASK AND THE SECOND WORD OF THE DCT IN AC0.
PLACE THE CONTENTS OF AC0 IN THE CURRENT INTERRUPT MASK (LOCATION 5).
DO A MASK OUT FROM AC0 AND ENABLE INTERRUPTS (DOBS 0,CPU).
PLACE ADDRESS OF DEVICE INTERRUPT ROUTINE IN PROGRAM COUNTER.
STACK OVERFLOW?
YES
NO
ALL MODES
CONTINUE SEQUENTIAL OPERATION WITH THE WORD ADDRESSED BY THE PROGRAM COUNTER.
TRANSFER CONTROL TO STACK FAULT ROUTINE
END OF VCT INSTRUCTION
DG-00570

Mode B - Part I

Mode B is used if the direct bit is 1 and the other two control bits are 0. The address in the vector table is now used as the location 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 is used to construct the new interrupt priority mask, and succeeding words (if any) contain information to be used by the device interrupt handler.

Mode C - Part I

If the direct bit and push bit are both 1, and the stack change bit is 0, mode C is used. The mode B functions are performed, and in addition, a standard 5-word return block is pushed onto the stack. This block consists of the contents of the 4 accumulators, the carry bit, and the contents of physical location 0 (the program counter return value).

Mode D - Part I

Mode D is used if the direct bit and the stack change bits are 1 and the push bit is 0. The mode B functions are performed, and in addition, a new stack is set up for the interrupt handler and the old contents of physical locations $40\text{-}43_8$ (the user stack control words) are pushed onto the new stack.

Mode E - Part I

Mode E combines the functions of modes C and D. That is, the functions of mode B are performed, a new stack is set up, and a 5-word return block and the old stack control words are pushed onto the (new) stack.

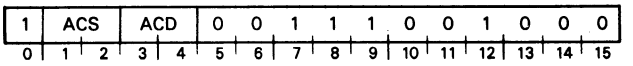
Modes B through E - Part II

Modes B through E use the same procedure for the remainder of the *Vector* instruction. The current priority mask is pushed onto the stack. A *Mask Out* instruction is then performed, using the logical OR of the current mask and the second word of the DCT. The Interrupt On flag is set to 1 and control passes 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.

Exchange Accumulators

XCH *acs,acd*

Valid for: CPU - IOP



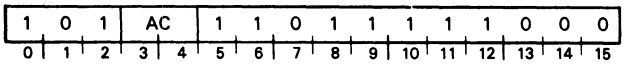
Exchanges the contents of two accumulators.

Places the original contents of ACS in ACD and the original contents of ACD in ACS.

Execute

XCT *ac*

Valid for: CPU - IOP



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 interruptable. 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 the programmer 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.*

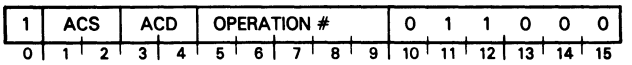
The results of XCT *are undefined if the specified accumulator contains an instruction that modifies that same accumulator. For example:*

```
      LDA   0,TOT
      XCT   0       ;UNDEFINED
      JMP   ON
TOT:  ADD   1,0
```

Extended Operation

XOP *acs,acd,operation #*

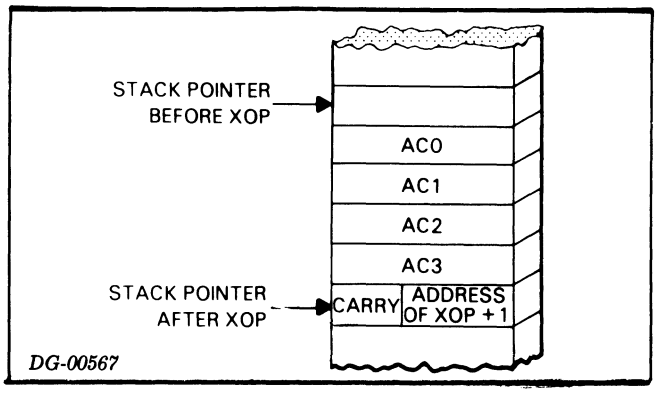
Valid for: CPU - IOP



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:

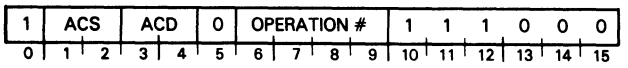


This return block is configured so that the XOP procedure can return control to the calling program via the POP BLOCK instruction.

Alternate Extended Operation

XOP1 *acs,acd,operation #*

Valid for: CPU - IOP

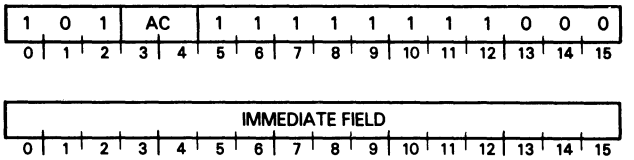


This instruction operates exactly like the *Extended Operation* instruction except that it adds 32_{10} to the entry number before it adds the entry number to the XOP origin address. In addition, it can specify only 16 entry locations.

Exclusive OR Immediate

XORI *i,ac*

Valid for: CPU - IOP

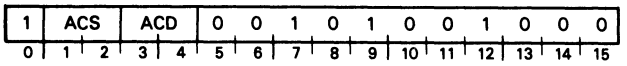


Forms the logical exclusive OR of the contents of the immediate field and the contents of the specified AC and places the result in the specified AC.

Exclusive OR

XOR *acs,acd*

Valid for: CPU - IOP



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.

This page intentionally left blank.

Chapter VI
M/600 I/O INSTRUCTIONS

Chapter VI lists the M/600 I/O instructions intended for a specific device such as the MAP, the BMC, and special CPU instructions. 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
- a functional description of each instruction

Some instructions can only be executed by the host processor, while others can also be executed by the I/O processor and/or the Data Control Unit. A label with each instruction indicates which processors can execute that 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.

Many of these instructions have special mnemonics which can be used in place of the standard mnemonics. The one limitation is that the mnemonics for controlling the state of the Interrupt On flag cannot be appended to the 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. The special mnemonics are given first below, followed by the standard form.

CODING AIDS

We use certain conventions and abbreviations throughout this chapter to help you properly coding 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:

f	Device Flag Command
AC	Accumulator

BURST MULTIPLEXOR CHANNEL

Device Code - 5_8(Primary)

Priority Mask Bit - None

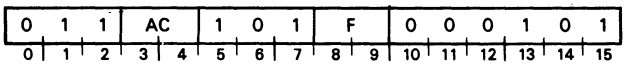
Device Flag Commands

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.

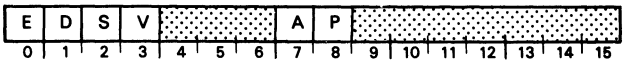
Read Status

DIC *[f]* *ac*,**BMC**

Valid for: CPU



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.

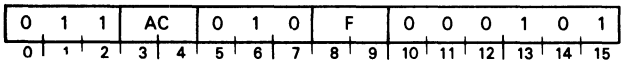


BITS	NAME	CONTENTS or FUNCTION
0	E	When 1, the channel has detected a validity protect error, an address parity error, or a data parity error.
1	D	When 1, the direction for a map data transfer is from the register(s) to memory (dump).
2	S	When 1, the channel is in two step diagnostic mode.
3	V	When 1, 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.

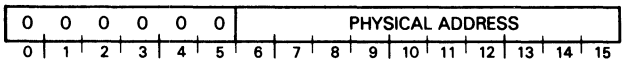
Specify Low-Order Address

DOA*[f]* *ac*,BMC

Valid for: CPU



The contents of the specified accumulator specify the low-order 10 bits of the 21-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.

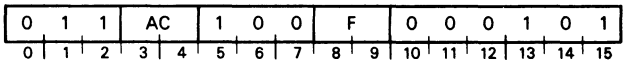


BITS	NAME	CONTENTS or FUNCTION
0-5	---	Must be 0.
6-15	LO ADDR	Specify the least significant bits of the physical address for the start of a map data transfer.

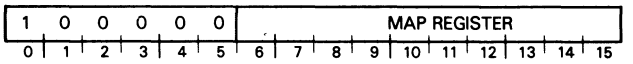
Specify Initial Map Register

DOB*[f]* *ac*,BMC

Valid for: CPU



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.

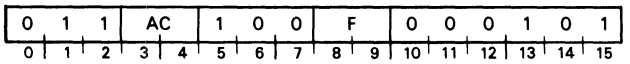


BITS	NAME	CONTENTS or FUNCTION
0	---	Must be 1.
1-5	---	Must be 0.
6-15	MAP REGISTER	Specify a map register as the first location for a map load/dump.

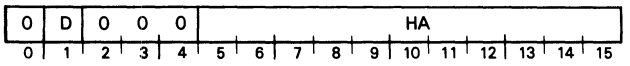
Specify High-Order Address

DOB*[f]* *ac*,BMC

Valid for: CPU



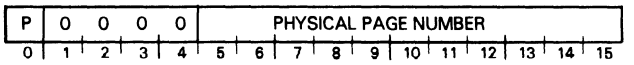
The contents of the specified accumulator determine the direction of the next map data transfer, as well as the high-order part of the physical memory address to be used. Bit 1 specifies whether map registers are to be loaded or dumped. Bits 5-15 are the the high-order 11 bits of the 21-bit physical 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.
2-4	---	Must be 0.
5-15	HI ADDR	Specify the most significant bits of the physical address for the start of the map data transfer.

Map Load Formats

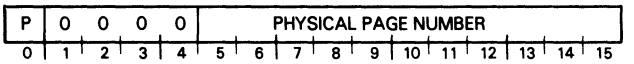
To load the map, the burst multiplexor transfers the contents of a memory buffer to the map register(s). The format of each word in the memory buffer is:



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.
1-4	---	Must be 0.
5-15	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;

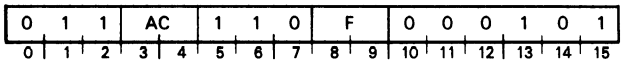


BITS	NAME	CONTENTS or FUNCTION
0	PROT	When 1, the channel cannot transfer data to/from memory in the specified physical page.
1-4	---	Reserved for future use.
5-15	PPN	Physical page number.

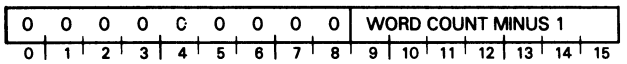
Specify Word Count

DOC*[f]* *ac*,BMC

Valid for: CPU



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.

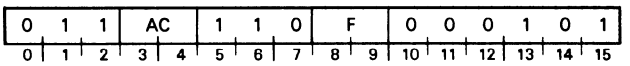


BITS	NAME	CONTENTS or FUNCTION
0-8	---	Must be 0.
9-15	COUNT	Specify a number that is one less less than the number of map registers to be loaded/dumped.

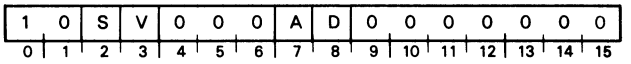
Set Status

DOC*[f]* *ac*,BMC

Valid for: CPU



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	STEP	When 1, the channel enters two-step diagnostic mode.
3	VPE	When 1, the channel forces a validity protect error.
4-6	---	Must be 0.
7	APE	When 1, the channel forces an address parity error.
8	DPE	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. 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.

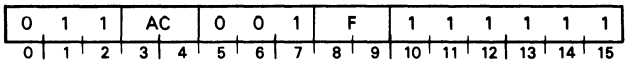
IORST Sets the interrupt enable flag to 0.

Read Switches

READS *ac*

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

Valid for: CPU - IOP



Places the contents of the console switches into an accumulator.

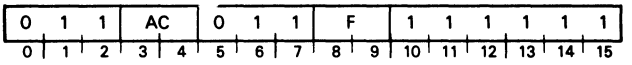
Places the setting of the console data switches in the specified accumulator. After the transfer, sets the Interrupt On flag according to the function specified by F.

Interrupt Acknowledge

INTA

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

Valid for: CPU - IOP



Returns device code of an interrupting device.

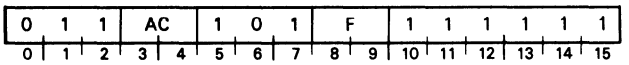
Places the six-bit device code of that device requesting an interrupt which is physically closest to the CPU on the I/O bus in bits 10-15 of the specified accumulator; sets bits 0-9 to 0. After the transfer, sets the Interrupt On flag according to the function specified by F.

Reset

IORST

DIC *[f]* *ac*,CPU

Valid for: CPU - IOP



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

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

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

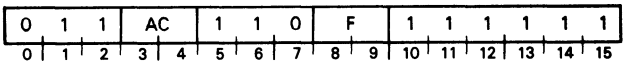
If the mnemonic **DIC** *is used to perform this function, you must code an accumulator to avoid assembly errors. During execution, the accumulator field is ignored and the contents of the accumulator remain unchanged.*

Halt

HALTA *ac*

DOC *[f]* *ac*,CPU

Valid for: CPU - IOP



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.

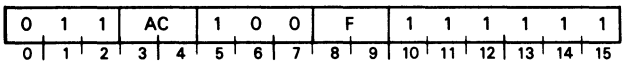
NOTE: *The assembler recognizes the mnemonic* **HALT** *as equivalent to the instruction* **HALTA 0.**

Mask Out

MSKO

DOB *[f]* *ac*,CPU

Valid for: CPU - IOP



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.

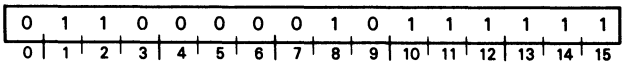
NOTE: *A 1 in any bit disables interrupt requests at devices which use that bit as a mask.*

Interrupt Disable

INTDS

NIOC CPU

Valid for: CPU - IOP



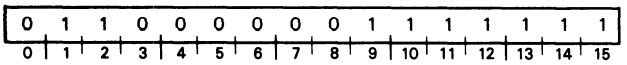
Sets Interrupt On flag to 0.

Interrupt Enable

INTEN

NIOS CPU

Valid for: CPU - IOP



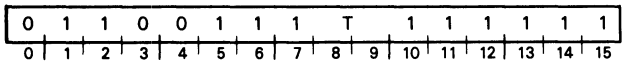
Sets Interrupt On flag to 1.

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 interruptable, then interrupts can occur as soon as the instruction begins to execute.

CPU Skip

SKP *[t]*CPU

Valid for: CPU - IOP



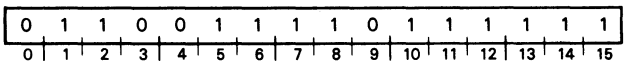
If the test condition specified by T is true, the next sequential word is skipped.

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

CPU Skip If Power Fail Flag Is One

SKPDN CPU

Valid for: CPU - IOP

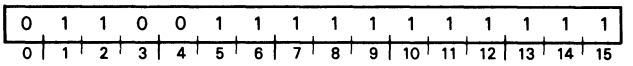


If the Power Fail flag is 1 (i.e., power is failing), the instruction skips the next sequential word.

CPU Skip If Power Fail Flag Is Zero

SKPDZ CPU

Valid for: CPU - IOP



If the Power Fail flag is 0 (i.e., power is not failing), the instruction skips the next sequential word.

HOST/DCU COMMUNICATION

Device Code, DCU from Host - 34_8(Primary)

Device Code, Host from DCU and local I/O devices - 76_8 (primary)

Priority Mask Bit, DCU from Host - 4

Priority Mask Bit, Host from DCU and local I/O devices - 4

Device Flag Commands

There are three cross interrupt/status flags which are used in Host/DCU communication. The Host has a Done flag which can be tested by the DCU. the DCU has both a Busy and Done flag which can be tested by the Host. The device flag commands for the five Host instructions control these Busy and Done flags as follows:

f=S Sets the Host Done flag to 1 and initiates an interrupt request to the DCU.

f=C Sets the DCU Done flag to 0.

f=P If the DCU is in diagnostic mode, steps the DCU internal clock. If the DCU is not in diagnostic mode, has no effect.

IORST Sets the DCU Done flag to 0, causes the DCU to execute an I/O Reset instruction, and places the DCU in diagnostic mode.

The device flag commands for the three DCU instructions control the Busy and Done flags as follows:

f=S Sets the DCU Done flag to 1 and initiates an interrupt request to the Host.

f=C Sets the Host Done flag to 0.

f=P Sets the DCU Real Time Clock flag to 1.

IORST Turns off the DCU Real Time Clock, and sets the Host Done flag to zero.

NOTES: *The DCU Done flag is read as Done by the Host. The Host Done flag is read as Done by the DCU.*

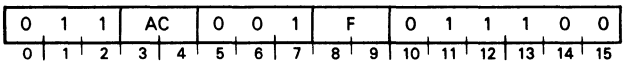
A DCU Run flag is read as busy by the Host. It is set to 1 when the DCU is not in a halted state.

The DCU Real Time Clock flag is read as Busy by the DCU. It is set to 1 when the Real Time Clock completes one cycle. When set to 1, it initiates an interrupt request to the DCU. The Host Done flag can also initiate an interrupt request from the same device code.

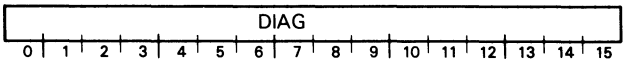
Read Diagnostic Data

DIA*[f]* *ac*,DCU

Valid for: CPU - IOP



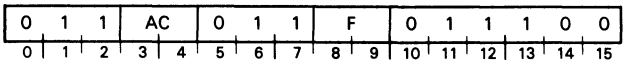
Loads diagnostic data for the data control unit into bits 0-15 of the specified accumulator. Sixteen signals provide the diagnostic data, with 0 signifying a low signal and 1 a high signal. Performs the function specified by F after the transfer. The format of the specified accumulator is as follows:



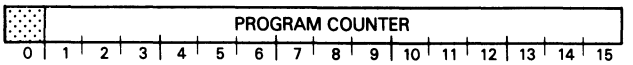
Read Program Counter

DIB*[f]* *ac*,DCU

Valid for: CPU - IOP



Loads the contents of the data control unit's program counter into bits 1-15 of the specified accumulator if the processor is stopped. If it is running, the results are unpredictable. The setting of bit 0 of the AC is undefined in either case. The previous contents of the specified AC are lost. After the transfer, the instruction performs the function specified by F. The format of the specified AC is as follows:

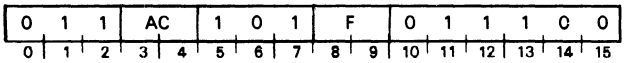


BITS	NAME	CONTENTS or FUNCTION
0	---	Reserved for future use
1-15	Program Counter	Present contents of the DCU's program counter

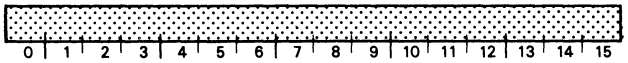
Reset

DIC*[f]* *ac*,DCU

Valid for: CPU - IOP



Stops the data control unit, places it in diagnostic mode, and forces it to simulate an *I/O Reset* instruction to reset all of the I/O devices attached to its I/O bus. Disables interrupts from the data control unit to the host computer, and sets the Host and DCU Done flags to 0. Sets bits 0-15 of the specified accumulator to 0. After the reset operation has been completed, the instruction performs the function specified by F. The format of the specified AC is as follows:

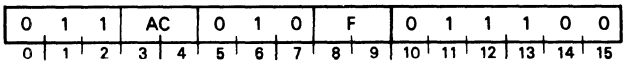


BITS	NAME	CONTENTS or FUNCTION
0-15	---	Reserved for future use.

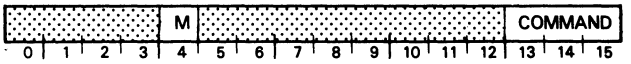
Control Mode

DOA*[f]* *ac*,DCU

Valid for: CPU - IOP



Uses bit 4 of the specified AC to set the mode of the data control unit. If bit 4 is 1, the instruction places the processor in diagnostic mode with its internal clock stopped. If bit 4 is 0, the instruction places the processor in normal operating mode with its internal clock running. Bits 13-15 cause the processor to stop, start, or continue. The stop command halts the processor at the end of the current instruction. The start command loads the program counter of the data control unit from the lower 15 bits of the HTDCU register before execution begins. The continue command begins execution of the DCU processor at the location contained in its program counter. Both the start command and the continue command are ignored if the processor is already running. The instruction ignores bits 0-3 and bits 5-12 of the accumulator. After the state of the processor has been set, the instruction performs the function specified by F. The contents of the accumulator remain unchanged and its format is as follows:

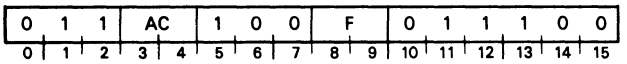


BITS	NAME	CONTENTS or FUNCTION
0-3	---	Reserved for future use.
4	MODE	Set the operating mode as follows: 0 Normal mode 1 Diagnostic mode
5-12	---	Reserved for future use.
13-15	COMMAND	Control the processor as follows: 100 Stop 001 Continue 010 Start at location in HTDCU register 000 011 101 Reserved for 110 future use. 111

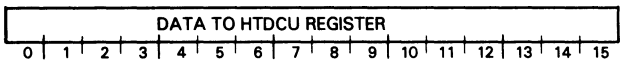
Load HTDCU Register

DOB*[f]* *ac*,DCU

Valid for: CPU - IOP



Loads bits 0-15 of the specified accumulator into the data control unit's HTDCU register. After the data transfer, the instruction performs the function specified by F. The contents of the specified AC remain unchanged. The format of the specified AC is as follows:

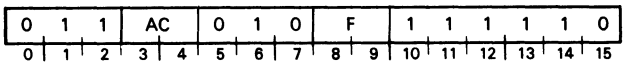


BITS	NAME	CONTENTS or FUNCTION
0-15	HTDCU Data	Data to be placed in the HTDCU register of the data control unit.

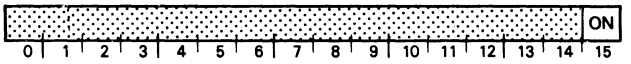
Control Real-Time Clock

DOA*[f]* *ac*,DCUI

Valid for: CPU - IOP



Uses bit 15 of the specified accumulator to control the real-time clock. If bit 15 is 0, the instruction turns the real-time clock off. If bit 15 is 1, the instruction turns the real-time clock on. The instruction sets the RTC flag to 0 any time within the first clock period after the clock is turned on. Ignores Bits 0-14 of the accumulator. After the real-time clock is turned on or off, the instruction performs the function specified by F. The contents of the AC remain unchanged. The format of the AC is as follows:

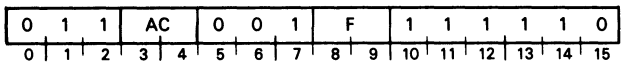


BITS	NAME	CONTENTS or FUNCTION
0-14	---	Reserved for future use.
15	On	Control the real-time clock as follows: 0 Turn off 1 Turn on

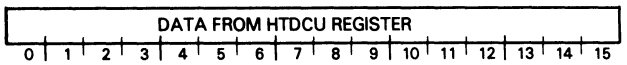
Read HTDCU Register

DIA*[f]* *ac*,DCUI

Valid for: CPU - IOP



Loads the contents of the data control unit's HTDCU register into bits 0-15 of the specified accumulator. After the data transfer, the instruction performs the function specified by F. The format of the AC is as follows:

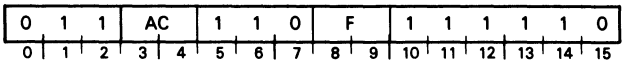


BITS	NAME	CONTENTS or FUNCTION
0-15	HTDCU Data	Present contents of the HTDCU register of the data control unit.

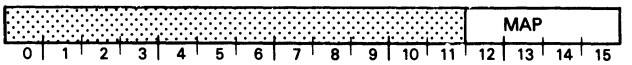
Select Data Channel Map

DOC*[f]* *ac*,DCUI

Valid for: CPU - IOP



Bits 12-15 of the specified accumulator select the data channel map which is to be used to map logical addresses from the data control unit into physical addresses in the host memory. The default value of the select bits for the data channel map is 0000. Bits 0-11 of the AC are ignored. After the data channel map has been selected, the function specified by F is performed. The contents of the AC remain unchanged. The format of the AC is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-11	---	Reserved for future use.
12-15	Map	Select the data channel map as follows: 0000 Data channel map A 0001 Data channel map B 0010 Data channel map C 0100 Data channel map D

DEMAND PAGING

Device Code - 4_8(Primary)

Priority Mask Bit - None

Device Flag Commands

f=S No effect.

f=C Sets the Page-use flags in the selected table to 0.

f=P No effect.

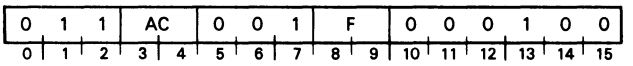
IORST Disables demand paging.

NOTE: *When demand paging is enabled, the auto-increment and auto-decrement feature is suppressed. Indirection chains passing through any of locations 20_8 -37_8 proceed without altering that location.*

Read Page-use Flags

DIA *[f]* *ac*,**DMP**

Valid for: CPU

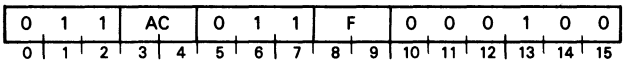


Loads the specified accumulator with the flags selected by the previous *Select Page-Use Table and Word* instruction. Each flag in the "referenced" or "modified" tables is set to 1 if the corresponding memory page has been referenced or modified since the last time the table was cleared.

Read Breakpoint Control Flags

DIB *[f]* *ac*,**DMP**

Valid for: CPU

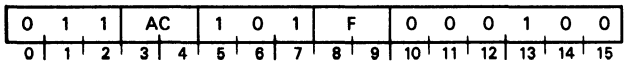


Loads bits 8-15 of the specified accumulator with the breakpoint control flags; set bits 0-7 to 0.

Read Breakpoint Address

DIC*[f]* *ac*,DMP

Valid for: CPU

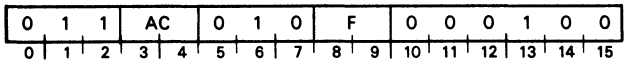


Loads bit 1-15 of the specified accumulator with the contents of the breakpoint address register; sets bit 0 of the accumulator to zero.

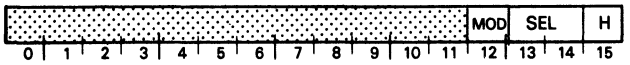
Select Page-Use Table and Word

DOA*[f]* *ac*,DMP

Valid for: CPU



Uses the contents of the specified accumulator to select one word of the reference or modified tables for reading or clearing. The accumulator format is given below.

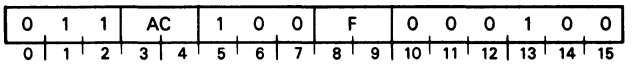


BITS	NAME	CONTENTS or FUNCTION
0-11	---	Reserved for future use.
12	MOD	0 selects reference table. 1 selects modified table.
13,14	SEL	00 selects tables for unmapped address space. 01 selects tables for user map B. 10 selects tables for user map A. 11 reserved for future use.
15	HOB	0 selects least significant bits pages 0-15 1 selects most significant bits pages 16-31

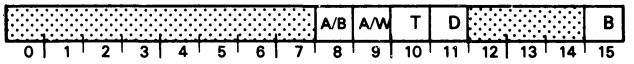
Set Breakpoint Control Flags

DOB*[f]* *ac*,DMP

Valid for: CPU



Sets the breakpoint control flags as determined by the contents of the specified accumulator. The format of the accumulator is given below:



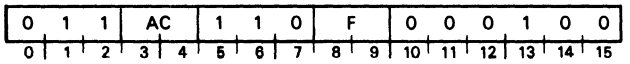
BITS	NAME	CONTENTS or FUNCTION
0-7	---	Reserved for future use
8	A/B	0 sets breakpoint in map A 1 sets breakpoint in map B
9	A/W	0 specifies Address trap 1 specifies Write trap
10	T	0 Turns off Trace trap 1 Turns on Trace trap
11[1]	D	0 turns off demand paging 1 turns on demand paging
12-14	---	Reserved for future use
15	B	0 enables breakpoint facility 1 disables breakpoint facility

[1] *When demand paging is enabled, the auto-increment and auto-decrement feature is suppressed for CPU's Map A and Map B logical address spaces. When demand paging is enabled, indirection chains within CPU Map A and Map B logical address spaces pass through locations 20₈ -37₈ without altering that location.*

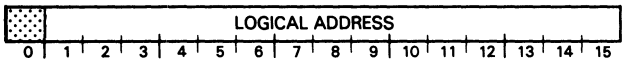
Set Breakpoint Address

DOC*[f]* *ac*,DMP

Valid for: CPU



Loads the breakpoint address register from the specified accumulator. The address format is given below.



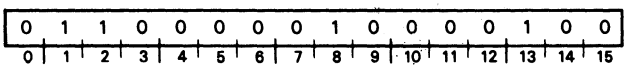
BITS	NAME	CONTENTS or FUNCTION
0	---	Reserved (see note.)
1-15	LOGICAL	Logical address for Address or Write traps.

NOTE: *Bit 0 of the breakpoint address word* in the context block *is used to indicate that a floating point instruction was in progress when the fault occurred. This bit must be preserved in the context block; however, it has no function in this instruction.*

Clear Page-use Flags

NIOC *ac*,DMP

Valid for: CPU



Sets to 0 all 16 flags specified by the previously issued *Select Table and Word* instruction.

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.

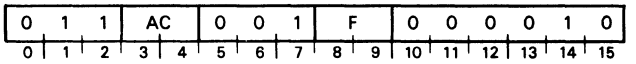
f=P No effect.

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.

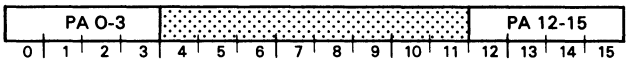
Read Memory Fault Address

DIA *[f]* *ac*,**ERCC**

Valid for: CPU



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 X4/0-3 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:

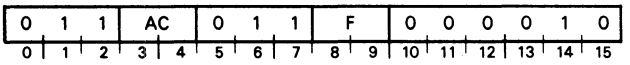


BITS	NAME	CONTENTS or FUNCTION
0-3	PA 0-3	Complement of bits X4/0-3 of the physical address of the memory location in error.
4-11	---	Reserved for future use.
12-15	PA 12-15	Complement of bits 12-15 of the physical address of the memory location in error.

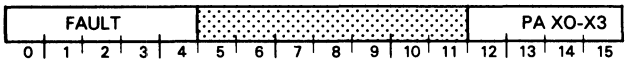
Read Memory Fault Code

DIB*[f]* *ac*,ERCC

Valid for: CPU



Places a 5-bit error code in bits 0-4 of the specified accumulator. This code identifies the bit in error that was corrected. Sets bits 5-11 of the accumulator to 0 and places the complement of the four high-order bits (X0-X3) of the physical address of the failing location in bits 12-15. The accumulator format is as follows:

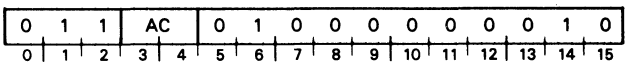


BITS	NAME	CONTENTS or FUNCTION
0-4	Code	A 5-bit code identifying which bit has an error 00000 No error 00001 Check bit 4 00010 Check bit 3 00011 Data bit 0 00100 Check bit 2 00101 Data bit 1 00110 Multiple bit error 00111 Data bit 3 01000 Check bit 1 01001 Data bit 4 01010 All 21 bits in memory are 1 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	Complements of bits X0-X3 of the physical address of the memory location in error.

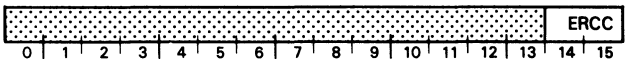
Enable ERCC

DOA*[f]* *ac*,ERCC

Valid for: CPU



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/IOP COMMUNICATION

Device Code, Host to IOP - 65_8(Primary)

Device Code, IOP to Host and local I/O - 4_8(Primary)

Priority Mask Bit, Host to IOP - 5 (Primary)

Priority Mask Bit, IOP to Host - 5 (Primary)

Priority Mask Bit, IOP Timer - 2

Priority Mask Bit, IOP Map - None

Device Flag Commands

There are two Busy flags and two Done flags associated with Host/IOP communication. The Host has a Busy and a Done flag which can be tested by the IOP. The IOP also has a Busy and a Done flag which can be tested by the Host. The device flag commands for the five Host instructions control these Busy and Done flags as follows:

f=S Sets the Host Busy and the IOP interrupt request flags to 1.

f=C Sets the IOP Done and Host interrupt request flags to 0.

f=P Sets the IOP parity error and Host interrupt request flags to 0.

IORST Sets the IOP Done flag, Host interrupt request flag, micro-interrupt request flag, and Host interrupt mask bit to 0, also resets the IOP processor and its I/O devices.

The device flag commands for the four IOP instructions control the Busy and Done flags as follows:

f=S Sets the IOP Done and Host interrupt request flags to 1.

f=C Sets the Host Busy and IOP interrupt request flags to 0.

f=P Sets the Host Done and IOP interrupt request flags to 0.

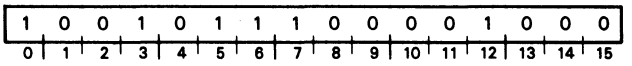
IORST Sets bits 2-4, 14 and 15 of the map status and parity control register, Host Done flag, IOP interrupt request flag, and IOP interrupt mask bits to 0.

NOTE: *An IOP Run flag is read as Busy by the Host. It is set to 1 when the IOP is not in a halted state.*

Load Map

LMP

Valid for: CPU



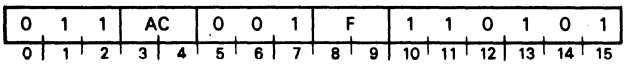
Load a number of map entries from a table in memory. AC2 must contain the starting address of the table. AC1 must contain the number of entries to load. AC0 and AC3 are unused. The format of each entry to be loaded into the IOP map is shown below.

BITS	NAME	FUNCTION
0	HOST	0 = map page into local memory. 1 = map page into host memory.
1-5	PAGE	Logical page number.
6-14	---	Reserved for future use.
15	DCH LD	0 = load entry into USER map. 1 = load entry into DCH map.

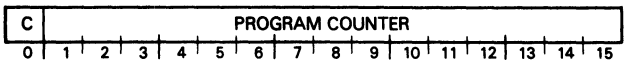
Read PC Save Register

DIA*[f]* *ac*,IOP

Valid for: CPU



Loads the contents of the IOP's PC Save register into the specified accumulator. The IOP updates the PC Save register each time the IOP halts. The format of the accumulator is as follows.

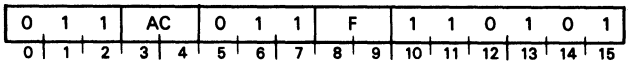


BITS	NAME	CONTENTS or FUNCTION
0	C	IOP Carry bit
1-15	PC	IOP Program counter

Read Console Buffer

DIB*[f]* *ac*,IOP

Valid for: CPU

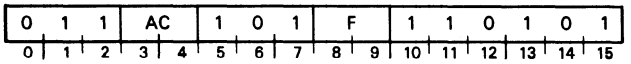


Loads the contents of the IOP console buffer into the specified accumulator. The console buffer contains the result of the last console operation performed by the IOP (such as *Examine*).

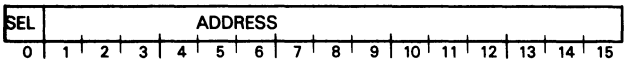
Read Address Buffer

DIC*[f]* *ac*,IOP

Valid for: CPU



Loads the contents of the IOP address buffer into the specified accumulator. The IOP address buffer contains the address of the last Host memory reference. If the address save bit in the IOP 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.

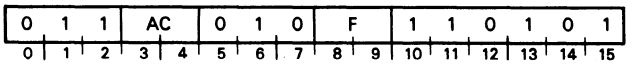


BITS	NAME	CONTENTS or FUNCTION
0	SEL	Current value of the least significant host data channel map select bit 0 DCH map A or C 1 DCH map B or D
1-15	---	IOP Logical Address

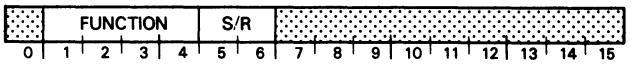
Control Console Function Register

DOA*[f]* *ac*,IOP

Valid for: CPU



Stores the contents of the specified accumulator into the IOP console function register. The format of the specified accumulator is:



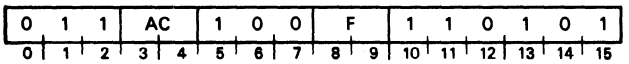
BITS	NAME	CONTENTS or FUNCTION
0	---	Reserved for future use
1-4	FUNCTION	Selects action when bits 5 & 6 = 0: 0000 Examine AC0 0001 Examine AC1 0010 Examine AC2 0011 Examine AC3 0100 Deposit AC0 0101 Deposit AC1 0110 Deposit AC2 0111 Deposit AC3 1000 Deposit 1001 Deposit Next 1010 Examine 1011 Examine 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 IOP ignores functions other than* **RESET**, **STOP**, **EXAMINE**, *and* **INST** *when it is running. The STOP function halts the IOP* after *it completes the instruction currently executing.*

Control Switch Register

DOB*[f]* *ac*,IOP

Valid for: CPU

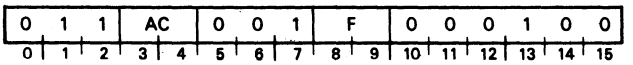


Stores the contents of the specified accumulator into the IOP switch register. The IOP switch register contains the address or data for the IOP console operations.

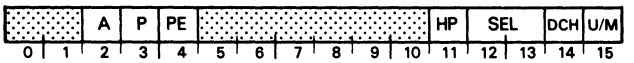
IOP Read Map Status and Parity Control

DIA*[f]* *ac*,IOPI

Valid for: - IOP



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:

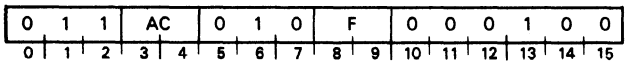


BITS	NAME	CONTENTS or FUNCTION
0,1	---	Reserved for future use.
2	A	Address Save on
3	P	Parity Test on
3	PE	Parity Checking on
5-10	---	Reserved for future use.
11	HP	Bit for MAP page selected by previous DOA: 0 = page is mapped into IOP local memory. 1 = page is mapped into host memory.
12,13	SEL	Currently 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.

IOP Control (and Select) Map and Page/Parity

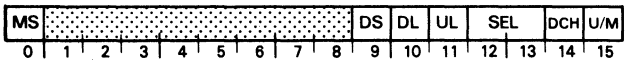
DOA*[f]* *ac*,IOPI

Valid for: - IOP

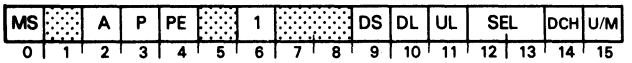


Stores the specified accumulator into the map status and parity control register as summarized below. Depending on the contents of bit 0 in the specified accumulator, the contents of bits 12-15 may be ignored; also, the contents of bit 6 controls the interpretation of bits 1-5. Consequently, the accumulator may take one of the following formats:

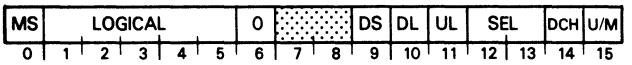
Format 1, Control Map Only



Format 2, Control Map and Parity



Format 3, Control Map and Select Page

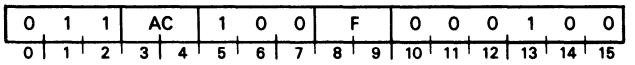


BITS	NAME	CONTENTS or FUNCTION
0	MSLE	Permit loading of map status (bits 12-15).
1	---	Used for map reading only.
2	A	Address Save on.
3	P	Parity Test on.
4	PE	Parity checking on.
1-5	LOGICAL	Selects Logical Page
6	PCL	Permit loading of parity control flags.
7,8	---	Reserved for future use.
9	DS	Suppress 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.

IOP Generate Micro-interrupt

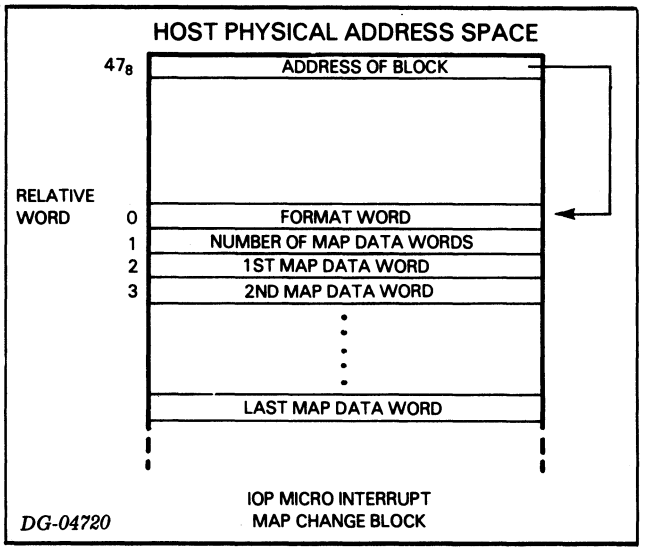
DOB*[f]* *ac*,IOPI

Valid for: - IOP



Causes the execution of a host map load which is invisible to the host program. Physical location 47_8 in host memory must contain a pointer to a block of data as shown in the accompanying diagram. When the map change is complete, the host program resumes execution and the IOP's Done flag for device 4 is set to 1.

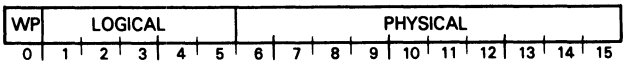
The accompanying diagram shows the format of the map change block.



Bits 6-8 of the format word select the map to be loaded as shown below. The other bits in the word are unused.

Bits	Which map to load
000	User A.
001	Reserved for future use.
010	User B.
011	Reserved for future use.
100	Data channel A.
101	Data channel B.
110	Data channel C.
111	Data channel D.

The format of each map data word is shown below.



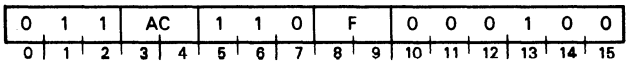
BITS	NAME	CONTENTS or FUNCTION
0	WP	Unused for data channel maps; write protect for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

NOTE: *The host must have interrupts enabled for a micro-interrupt to occur. Setting the host's interrupt mask bit 5 does not suppress micro-interrupts.*

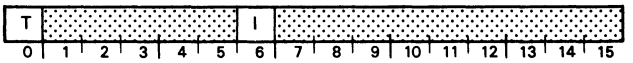
IOP Control Timer

DOC*[f]* *ac*,IOPI

Valid for: - IOP



Store the contents of the specified accumulator into the timer control register and perform the function determined by the bits as shown below.



BITS	NAME	CONTENTS or FUNCTION
0	TSTART	0 = Stop timer 1 = start timer.
1-5	---	Reserved for future use.
6	INTCLR	If 1, clear interrupt.
7-15	---	Reserved for future use.

NOTE: *The timer interrupt may be disabled without stopping the timer by setting bit 2 in the IOP interrupt mask register.*

MEMORY ALLOCATION AND PROTECTION

Device Code - 4_8(Primary)

Priority Mask Bit - None

Device Flag Commands

f=S No effect.

f=C No effect.

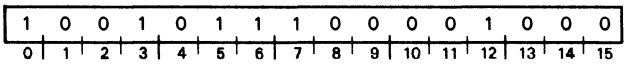
f=P Enables Map Single Cycle.

IORST Disables Map.

Load Map

LMP

Valid for: CPU



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.

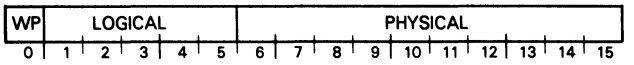
AC1 must contain an unsigned integer which is the number of words to be loaded into the MAP. Bits 1-15 of AC2 must contain the address of the first word to be loaded. If bit 0 of AC2 is 1, the instruction follows the indirection chain and places the resultant effective address in AC2. AC0 and AC3 are 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 interruptable 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:



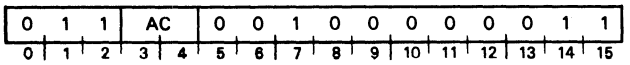
BITS	NAME	CONTENTS or FUNCTION
0	WP	Unused for data channel maps; write protect for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

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

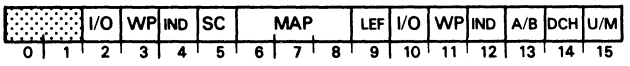
DIA *ac*,MAP

Valid for: CPU



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:

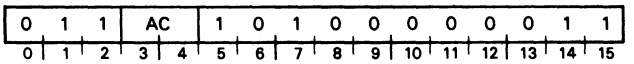


BITS	NAME	CONTENTS or FUNCTION
0-1	---	Reserved for future use.
2	I/O	If 1, the last protection fault was an I/O protection fault.
3	WP	If 1, the last protection fault was a write protection fault.
4	IND	If 1, the last protection fault was an indirect protection fault.
5	Single Cycle	If 1, the last map 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
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 the user map for user A. If 1, the last Load Map Status instruction enabled the user map for user 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

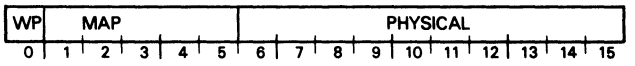
DIC *ac*,MAP

Valid for: CPU



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:

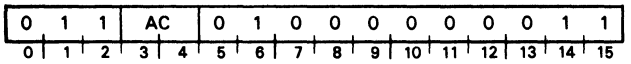


BITS	NAME	CONTENTS
0	WP	The write protect bit for the logical page which corresponds to the physical page specified by bits 6-15.
1-3	Map	The map which was used to perform the translation between logical page number and physical page number is as follows: 000 User A 001 Reserved for future use. 010 User 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

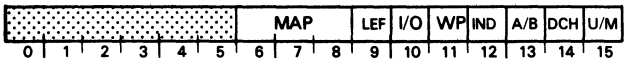
DOA *ac*,MAP

Valid for: CPU



Defines the parameters of a new map.

Places the contents of the specified AC are placed in the MAP status register. The contents of the specified AC remain unchanged. The format of the spcified AC is as follows:



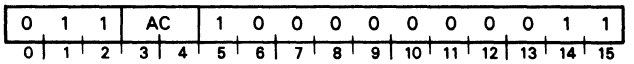
BITS	NAME	CONTENTS or FUNCTION
0-5	---	Reserved for future use.
6-8	MAP SEL	Specify which map will be loaded by the next Load Map instruction as follows: 000 User A 001 Reserved for future use 010 User B 011 Reserved for future use 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel 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, *or* Restore *instruction that does not cause a stack fault.*

Map Page 31

DOB *ac*,MAP

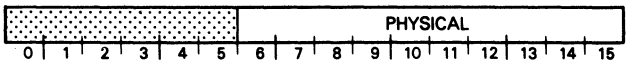
Valid for: CPU



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:

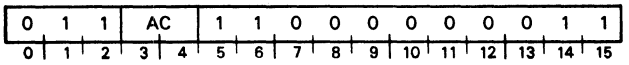


BITS	NAME	CONTENTS or FUNCTION
0-5	---	Reserved for future use.
6-15	Physical	The number of the physical page to which logical page 31 should be mapped when in unmapped mode.

Initiate Page Check

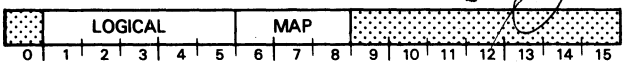
DOC *ac*,MAP

Valid for: CPU



Identifies a logical page. The *Page Check* instruction will find the corresponding physical page.

Transfers 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:



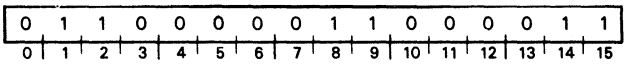
BITS	NAME	CONTENTS or FUNCTION
0	---	Reserved for future use.
1-5	Logical Page	Number of the logical block for which the check is requested.
6-8	Map	Specify which map should be used for the check as follows: 000 User A 001 Reserved for future use. 010 User B 011 Reserved for future use. 100 Data channel 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

Valid for: CPU



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 priveleged 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 again 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 - 53_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.

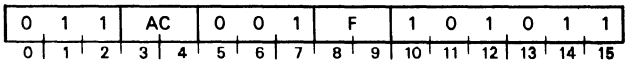
f=P No effect.

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.

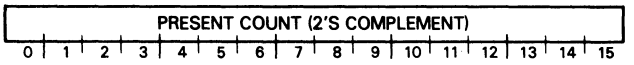
Read Count

DIA *[f]* *ac*,**PIT**

Valid for: CPU



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:

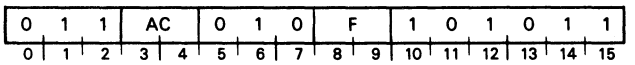


BITS	NAME	CONTENTS or FUNCTION
0-15	Count	Current value of the PIT counter within one count cycle.

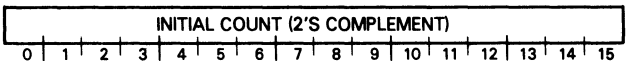
Specify Initial Count

DOA *[f]* *ac*,**PIT**

Valid for: CPU



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.

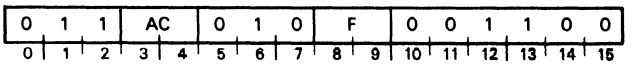
f=P No effect.

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.

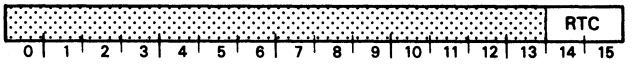
Select RTC Frequency

DOA*[f]* *ac*,**RTC**

Valid for: CPU



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.
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.

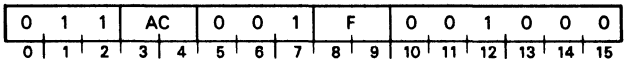
f=P No effect.

IORST Sets the Busy and Done flags, the interrupt request flag, and the interrupt mask bit (bit 14) to 0.

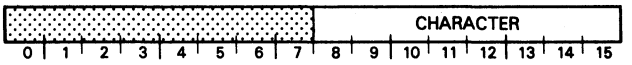
Read Character Buffer

DIA*[f]* *ac*,TTI

Valid for: CPU



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	Character	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.

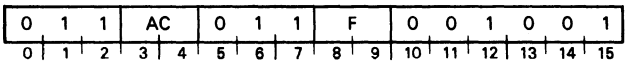
f=P No effect.

IORST Sets the Busy and Done flags, the interrupt request flag, and the interrupt mask bit (bit 15) to 0.

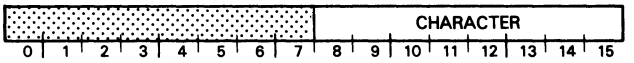
Load Character Buffer

DOA*[f] ac*,TTO

Valid for: CPU



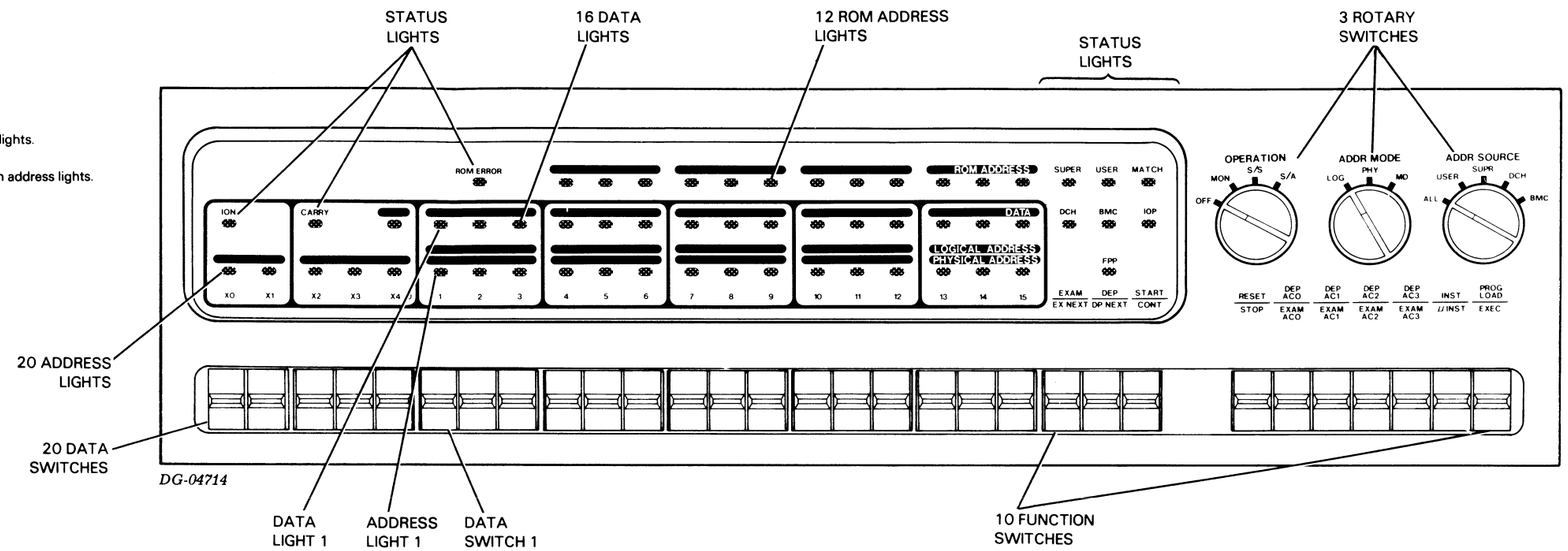
Loads bits 8-15 of the specified accumulator into the controller's output buffer. After the data transfer, sets the controller's Busy and Done flags according to the function specified by F. The contents of the specified accumulator remain unchanged. The format of the specified accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-7	----	Reserved for future use.
8-15	Character	The 8-bit character or 7-bit character with parity in bit position 8 to be placed in the output buffer.

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 addresses 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	Console monitors memory addressing by the burst multiplexor channel.

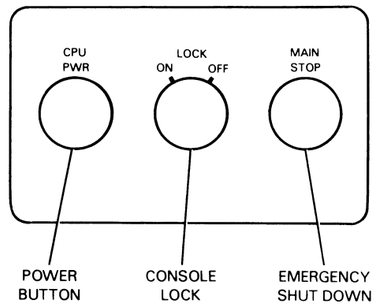


FUNCTION SWITCHES

NAME	POSITION	FUNCTION
Exam/	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.
Exam N	Down	Increments PC, and displays contents of that address.
Dep/	Up	Deposits value in data switches at PC address.
Dep N	Down	Increments PC and deposits value in data switches in that address.
St/	Up	Loads with value in data switches and starts normal execution. Also fills EA register.
Cont	Down	Initiates normal operation from the current state of the machine.
Rest/	Up	Resets CPU and issues an IORST. ROM address lights display 0002_8.
Stop	Down	Halts the CPU.
AC Dep/	Up	Loads the associated accumulator with the value in the data switches.
Exam	Down	Displays the contents of the associated accumulator.
Inst/	Up	Executes one machine instruction then halts the CPU.
U Inst	Down	Freezes the CPU after executing one microinstruction. Address lights show output of ALU.
P Load/	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.
Exec	Down	Executes instruction contained in data switches.

POWER PANEL

NAME	OPERATION
CPU PWR	Controls DC power to the ECLIPSE M/600 chassis
LOCK	Enables and disables console switches.
MAIN STOP	Shuts off all power to the cabinet when pushed.



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	The I/O processor is either normally executing or frozen, but not halted.
FPP	The floating point processor is performing a floating point operation.

* If a ROM parity occurs, the CPU freezes.

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.

Chapter VII
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 M/600 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 or 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.)

DG-04842

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	Specifies that the console will monitor memory 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

DG-04842

* 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.

DG-04843

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.

DG-04843

* 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 M/600 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.

* 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.)

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.

This page intentionally left blank.

APPENDIX A
STANDARD I/O DEVICE CODES

OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME	OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
00	----	--	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	DPM		Demand Paging Map	43	DIOT	6	Digital I/O timer
	IOPI	2,5	IOP Map, IDP Timer		PIT	11	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	9	Second fixed head disc
22	MTA	10	Magnetic tape	61	ADCV1	8	Second A/D converter
23	DACV	--	D/A converter	62	MTA1	10	Second magnetic tape
24	DCM	0	Data communications multiplexor	63	DACV1	--	Second D/A converter
25				64			
26	DKB	9	Fixed head DG/Disc	65	IOP I	5[5]	Host To IOP Interface
27	DPF	7	DG/Disc storage subsystem	66	DKB1	9	Second Fixed Head DG/Disc
30	QTY	14	Asynch. hardware multiplexor	67	DPF1	7	Second DG/Disc storage subsystem
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

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. Micro interrupts are not maskable

This page intentionally left blank.

APPENDIX B
OCTAL AND HEXADECIMAL CONVERSION

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

To convert a decimal number to octal or hexadecimal:

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

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

OCTAL CONVERSION TABLE

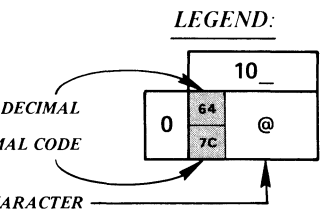
	8^5	8^4	8^3	8^2	8^1	8^0
0	0	0	0	0	0	0
1	32,768	4,096	512	64	8	1
2	65,536	8,192	1,024	128	16	2
3	98,304	12,228	1,536	192	24	3
4	131,072	16,384	2,048	256	32	4
5	163,840	20,480	2,560	320	40	5
6	196,608	24,576	3,072	384	48	6
7	229,376	28,672	3,584	448	56	7

HEXADECIMAL CONVERSION TABLE

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

This page intentionally left blank.

APPENDIX C
ASCII CHARACTER CODES



↥ *means* CONTROL

OCTAL	00_		01_		02_		03_		04_		05_		06_		07_	
0	0 00	NUL	8 16	BS *(BACK-SPACE)*	16 10	DLE ↥P	24 18	CAN ↥X	32 40	SPACE	40 4D	(	48 F0	Ø	56 F8	8
1	1 01	SOH ↥A	9 05	HT *(TAB)*	17 11	DC1 ↥Q	25 19	EM ↥Y	33 5A	!	41 5D	)	49 F1	1	57 F9	9
2	2 02	STX ↥B	10 15	NL *(NEW LINE)*	18 12	DC2 ↥R	26 3F	SUB ↥Z	34 7F	,, *(QUOTE)*	42 5C	*	50 F2	2	58 7A	:
3	3 03	ETX ↥C	11 0B	VT *(VERT. TAB)*	19 13	DC3 ↥S	27 27	ESC *(ESCAPE)*	35 7B	#	43 4E	+	51 F3	3	59 5E	;
4	4 37	EOT ↥D	12 06	FF *(FORM FEED)*	20 3C	DC4 ↥T	28 1C	FS ↥\	36 5B	$	44 6B	, *(COMMA)*	52 F4	4	60 4C	<
5	5 2D	ENQ ↥E	13 0D	RT *(RETURN)*	21 3D	NAK ↥U	29 1D	CS ↥]	37 6C	%	45 60	–	53 F5	5	61 7E	=
6	6 2E	ACK ↥F	14 0E	SO ↥N	22 32	SYN ↥V	30 1E	RS ↥↑	38 50	&	46 4B	. *(PERIOD)*	54 F6	6	62 6E	>
7	7 2F	BEL ↥G	15 0F	SI ↥O	23 26	ETB ↥W	31 1F	US ↥←	39 7D	, *(APOS)*	47 61	/	55 F7	7	63 6F	?

000 001 010 011 100 101 110 111

OCTAL	10_		11_		12_		13_		14_		15_		16_		17_	
0	64 7C	@	72 C8	H	80 D7	P	88 E7	X	96 79	` *(GRAVE)*	104 88	h	112 97	p	120 A7	x
1	65 C1	A	73 C9	I	81 D8	Q	89 E8	Y	97 81	a	105 89	i	113 98	q	121 A8	y
2	66 C2	B	74 D1	J	82 D9	R	90 E9	Z	98 82	b	106 91	j	114 99	r	122 A9	z
3	67 C3	C	75 D2	K	83 E2	S	91 8D	[	99 83	c	107 92	k	115 A2	s	123 C0	{
4	68 C4	D	76 D3	L	84 E3	T	92 E0	\	100 84	d	108 93	l	116 A3	t	124 4F	|
5	69 65	E	77 D4	M	85 E4	U	93 9D	]	101 85	e	109 94	m	117 A4	u	125 D0	}
6	70 C6	F	78 D5	N	86 E5	V	94 5F	↑ or ^	102 86	f	110 95	n	118 A5	v	126 A1	~ *(TILDE)*
7	71 C7	G	79 D6	O	87 E6	W	95 6D	← or_	103 87	g	111 96	o	119 A6	w	127 07	DEL *(RUBOUT)*

CHARACTER CODE IN OCTAL AT TOP AND LEFT OF CHARTS.

This page intentionally left blank.

APPENDIX D
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 E
COMPATIBILITY WITH NOVA LINE COMPUTERS

The ECLIPSE M/600 computers are compatible with Data General's NOVA line of computers. Any program presently running on any NOVA line computer will run on an ECLIPSE series computer without change provided that it does not violate any of the following constraints:

- The program may not be dependent on instruction execution times or Input/Output (I/O) transfer times. Times for the ECLIPSE series computers may be faster than a NOVA line computer depending upon the application.

- The program may not use any fixed-point arithmetic instructions that have both the *no-load* and *no-skip* options specified. The ECLIPSE series computers use these codes to implement instructions in the standard instruction set.

- The program may not require the hardware multiply/divide option available on any NOVA line computer.

- The program may not utilize the data channel increment or add-to-memory features.

- The program may not utilize either the memory management and protection option or the hardware floating point option currently available for NOVA line computers.

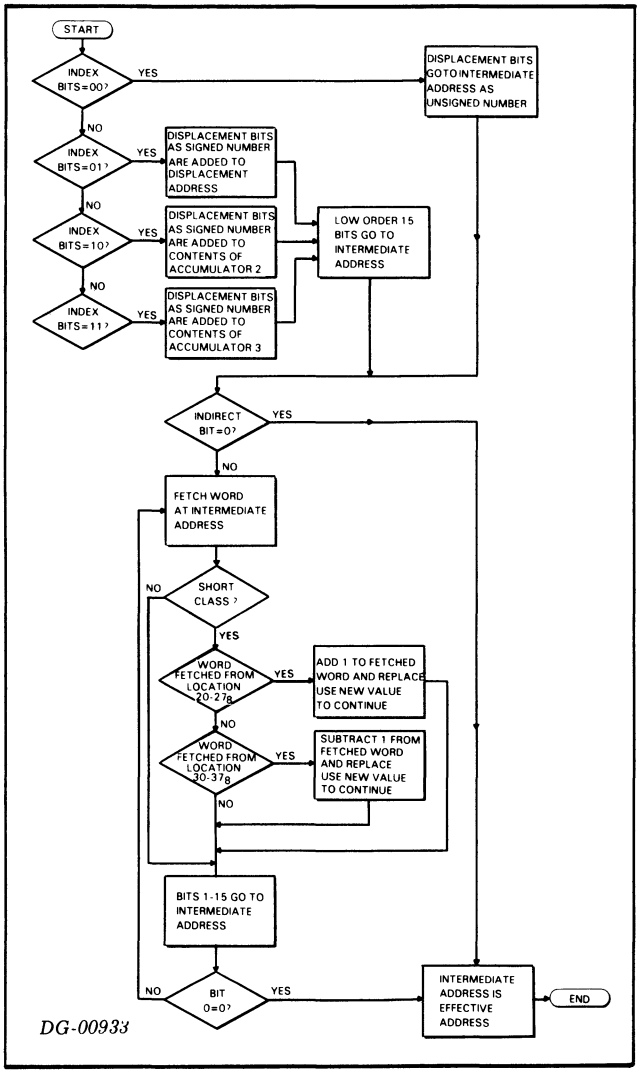
- The memory and I/O resources available on the ECLIPSE series computer should be at least equivalent to those available on the NOVA line computer for which the program was designed.

A violation of the third constraint can be easily corrected. The multiply and divide available in the ECLIPSE series computers standard instruction set are functionally equivalent to the operations provided in the hardware multiply/divided option for the NOVA line computers. Only the operation codes must be changed to take advantage of the ECLIPSE series computer's multiply and divide feature. Similarly, only small changes need be made to a program which uses the current NOVA line floating point option in order for that program to take advantage of the floating point option. The floating point number formats are the same.

This page intentionally left blank.

APPENDIX F
ADDRESSING

A flow diagram of the addressing process is shown below. See Chapter III for a detailed discussion of addressing.



This page intentionally left blank.

APPENDIX G
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
```

This page intentionally left blank.

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

This page intentionally left blank.

INSTRUCTION INDEX

* *EDIT Subinstruction*

** EDIT Subinstruction*

I/O INSTRUCTION INDEX

This page intentionally left blank.

DATA GENERAL DOMESTIC OFFICES

SALES AND SERVICE

ALABAMA

90 Bagby Drive
Birmingham, Alabama 35209
(205) 942-3730

ARIZONA

1950 E. Watkins
Phoenix, Arizona 85034
(602) 254-6412

Broadway Offices
Suite 100
4633 E. Broadway
Tucson, Arizona 85100
(602) 327-6811

ARKANSAS

Southland Plaza, Suite 308
6th and McKinley
Little Rock, Arkansas 72205
501-664-7075

CALIFORNIA

Data General Western
1136 North Gilbert Street B
Anaheim, California 92801
(714) 991-0130

Los Angeles Office
2221 Rosecrans Avenue
Suite 204
El Segundo, California 90245
(213) 973-0401

Suite 108
5110 Clinton Way
Fresno, California 93727
209-251-5027

2445 Faber Place
Palo Alto, California 94303
(415) 321-8010

4031 No. Freeway Blvd.
Sacramento, California 95834
(916) 920-3434

4305 Gesner Street
San Diego, California 92117
(714) 276-8450

120 Howard Street
San Francisco, California 94105
(415) 543-6730

1220 Village Way
Suite A
Santa Ana, California 92705
(714) 558-9421

5266 Hollister Avenue
Suite E
Santa Barbara, California 93111
(805) 967-5544

7011 Hayvenhurst Avenue
Suite A
Van Nuys, California 91406
(213) 787-8870

COLORADO

Denver Technological Center
Building No. 29
8455 E. Prentice Avenue
Englewood, Colorado 80110
(303) 773-2620

CONNECTICUT

50 Shaw Road
North Branford, Connecticut 06471
(203) 484-2771

FLORIDA

1001 Northwest 62nd Street
Suite 405
Ft. Lauderdale, Florida 33309
(305) 771-0784

6880 Lake Ellenor Drive
Suite 115
Orlando, Florida 32809
(305) 851-8230

5444 Bay Center Drive
Suite 707
Tampa, Florida 33605
(813) 879-2515

GEORGIA

1639 Tullie Circle
Suite 128
Atlanta, Georgia 30329
(404) 325-3181

IDAHO

130 South Cole Road
Boise, Idaho 83075
(208) 375-6510

ILLINOIS

2524 W. Farrelly Avenue
S.W. of Route 150 and Glen
Peoria, Illinois 61614
(309) 688-2224

Chicago Office
1111 Plaza Drive
Schaumburg, Illinois 60195
(312) 885-0505

INDIANA

2421 Production Drive
Suite 202
Indianapolis, Indiana 46241
(317) 248-8306

KENTUCKY

4299 Bardstown Rd.
Louisville, Kentucky 40218
(502) 491-6595

LOUISIANA

8211 Goodwood Blvd.
Suite E
Baton Rouge, Louisiana 70806
(504) 923-0660

MARYLAND

Washington D.C., Office
1900 Sulphur Spring Road
Suite 310
Baltimore, Maryland 21227
(301) 247-8900

MASSACHUSETTS

Data General Eastern
810 Memorial Drive
Cambridge, Massachusetts 02138
(617) 661-7710

1500 Main Street
Bay State West
Suite 1820
Springfield, Massachusetts 01103
(413) 739-9880

888 Worcester Road
Wellesley, Massachusetts 02181
(617) 235-8171

Mechanics Bank Tower
Suite 640
Worcester, Massachusetts 01608
(617) 754-3096

MICHIGAN

Detroit Office
1 Northland Drive Building
Suite 366
16900 West Eight Mile Road
Southfield, Michigan 48075
(313) 569-0200

MINNESOTA

Minneapolis Office
1600 East 78th Street
Richfield, Minnesota 55423
(612) 866-3005

MISSOURI

4520 Madison Avenue
Suite 210
Kansas City, Missouri 64111
(816) 531-1326

St. Louis Office
11854 Lakland Rd.
St. Louis, Missouri 63141
(314) 726-0811

NEVADA

3355 Spring Mountain Road
Suite 54
Las Vegas, Nevada 89100
(702) 876-0663

NEW HAMPSHIRE

491 Amherst Street
Nashua, New Hampshire 03060
(603) 889-8524

NEW JERSEY

1013 Annapolis Drive
Cherry Hill, New Jersey 08003

30 Galesi Drive
Wayne, New Jersey 07470
(201) 785-3910

NEW MEXICO

2601 Wyoming, N.E.
Suite A
Albuquerque, New Mexico 87112
(505) 294-1416

NEW YORK

3527 Harlem Road
Buffalo, New York 14200
(716) 835-3710

12 Avis Drive
Guptill Industrial Park Office Bldg.
Latham, New York 12110
(518) 783-3101

150 Broad Hollow Road
Third Floor
Melville, New York 11746
(516) 427-0311

L141 Meadowbrook Court
Newfield, New York 14867
(607) 272-9440

144 East 44th Street
New York, New York 10017
(212) 557-1122

625 Panorama Trail
Rochester, New York 14625
(716) 385-2000

104 Pickard Drive
Syracuse, New York 13211
(315) 455-1525

6 Corporate Park Drive
White Plains, New York 10604
(914) 694-1410

NORTH CAROLINA

2 Fairview Plaza
Suite 106
5950 Fairview Road
Charlotte, North Carolina 28210
(704) 554-8600

405 Parkway Drive
Greensboro, North Carolina 27401
(919) 378-9401

OHIO

Columbus Office
6161 Busch Blvd.
Suite 73
Columbus, Ohio 43229
(614) 882-1039

1161 Lyons Road
Building E
Dayton, Ohio 45459
(573) 435-1932

Cleveland Office
709 Brookpark Road
Brooklyn Heights, Ohio 44109
(216) 459-2300

OKLAHOMA

6161 North May Avenue
Suite 39W
Oklahoma City, Oklahoma 73112
(405) 840-3891

9726 E. 42nd South
Suite 200
Tulsa, Oklahoma 74145
(918) 664-8530

OREGON

4614 S.W. Kelly
Portland, Oregon 97201
(503) 223-7150

PENNSYLVANIA

Philadelphia Office
Dublin Hall, 4th Floor
1777 Walton Road
Blue Bell, Pennsylvania 19422
(215) 643-5515

Carnegie Office Park
600 Bell Avenue
Building 100
Carnegie, Pennsylvania 15106
(412) 279-3500

RHODE ISLAND

225 Newman Avenue
Rumford, Rhode Island 02916
(401) 438-0001

SOUTH CAROLINA

One Graystone West
240 Stoneridge Drive
Suite 413
Columbia, South Carolina 29210
(803) 798-6510

TENNESSEE

1515 East Magnolia Avenue
Hamilton East Building
Suite 310
Knoxville, Tennessee 37917
(615) 524-1841

1355 Lynnfield Road
Suite 221 Bldg. B
Memphis, Tennessee 38138
(901) 761-3366

TEXAS

111 West Anderson Lane
Suite 301
Austin, Texas 78752
(512) 458-4201

4450 Sigma Road
Suite 130
Dallas, Texas 75240
(214) 233-4496

1605 Beech Place
Suite B
El Paso, Texas 79925
(915) 778-6601

4949 W. 34th Street
Suite 123
Houston, Texas 77092
(713) 688-8641

5608 Malvey Avenue
Ft. Worth, Texas 76100
(817) 732-7075

UTAH

3644 West 2100 South
Salt Lake City, Utah 84120
(801) 972-1338

VIRGINIA

1757 Old Meadow Road
McLean, Virginia 22101
(703) 893-0910

5 Koger Executive Center
Suite 202
Norfolk, Virginia 23502
(804) 461-0933

5001 W. Broad Street
Suite 209
Richmond, Virginia 23230
(804) 288-1675

417 Apersen Road
Salem, Virginia 24153
(703) 389-8155

WASHINGTON

Seattle Office
Suite 107
10604 N.E. 38th Place
Quad. 1 North
Kirkland, Washington 98033
(206) 827-0503

WEST VIRGINIA

Union Building
Suite 400
Charleston, West Virginia 25301
(304) 345-2981

WISCONSIN

Milwaukee Office
10533 West National Avenue
Suite 300
West Allis, Wisconsin 53227
(414) 255-3230

DATA GENERAL INTERNATIONAL OFFICES

INTERNATIONAL OFFICES

ARGENTINA, PARAGUAY, BOLIVIA

Gases Industriales S.A.
Technica IC (Novadata)
5830 Callee Guatemala
2 Piso
Capital Federal
Buenos Aires, Argentina
37-5101, 37-5284, 37-8681,
37-8669, 37-8577
(33) 122421 AR INARG TLX

AUSTRALIA

Data General Australia Pty., Ltd.
30-32 Ellingworth Parade
Box Hill 3123
Melbourne, Victoria 3128
(03) 890633, (03) 890402
DAGEN 33041 TLX

Perth Office
43 Ventnor Avenue
West Perth, W. Australia 6000
092-215981
790-92093 TLX

Sydney Office
P.O. Box 117
40 Yeo Street
Neutral Bay Junction
N.S.W. 2089
(02) 9081366
790-25046 TLX

Brisbane Office
67 St. Paul's Terrace
Springhill, Queensland 4000
07-229-5942
790-40159 TLX

Adelaide Office
122-130 Carrington Street
Adelaide, 5000
South Australia
08-2237344

NEW ZEALAND

Data General New Zealand, Ltd.
Suite B, Level 13
William City Center
Plimmer Steps
Wellington, New Zealand
723-095
791-31002 TLX

Data General New Zealand, Ltd.
2nd Floor
Aetna Life Building
P.O. Box 5175
Wellesley Street
Auckland, New Zealand
33830, 33930
21793 TLX

AUSTRIA

Data General Gesellschaft M.B.H.
Perkorngasse 2
A-1100 Wien, Austria
(222) 73-4566/69
847-74559 TLX

BELGIUM

NV Data General, SA
Boulevard du Souverain 191-197
1160 Brussels, BTE 11
Belgium
(02) 660-49-44
61206 TLX

CANADA

415 Horner Avenue
Toronto, Ontario M8W 4W3
(416) 259-4271
610-492-2553 TWX

438 Isabay, Ste. □ 105
Ville St. Laurent, Quebec
(514) 342-5730
610-421-4758 TWX

210 Gladstone Avenue
Suite 2003
Ottawa, Ontario K29 OY6
(613) 563-1121
610-562-8536 TWX
053-3501 TLX

10250 Shellbridge Way
Richmond, B.C. V6X 2W7
(604) 273-2711
610-923-5080 TWX

5923 3rd Street S.E.
Calgary, Alberta
(403) 253-8427
610-821-2473 TWX

610 10339 - 124 Street
Edmonton, Alberta T5N 3W1
(403) 482-3443
610-831-1240 TWX

CHILE

Sistemas Y Equipos
De Computacion (SYNAL)
Avenida Andres Bello 1465
Santiago 9, Chile
SGO 260 TLX

COLOMBIA

MBB Companhia Ltda.
Carrera 7 N 81-26
APDO Aereo 90670
Bogata, Colombia
369-565

COSTA RICA

Sistemas Analiticos, SA
Apartado 6244
San Jose, Costa Rica
22-8156
(376) 2481 FEGOR TLX

DENMARK

Data General Denmark
Sjaellandsboen 2
2450 Kopenhavn SV
Denmark
(01) 305666
855-15827 TLX

ECUADOR

Sistemas Andinos S.A. (SANSA)
Av. Tarqui 809
5 Piso
Quito, Ecuador
520-957, 543-360
(308) 22295 PACOSA ED TLX

EGYPT & JORDAN

Trading & Engineering Assoc., S.A.
2 Maaruf Street
P.O. Box 1901
Cairo, Egypt
56665, 40007 Cable
927-2478 TEACOM UN. TLX

FINLAND & USSR

Oy Stromberg AB
P.O. Box 118
SF 00101 Helsinki 10
Finland
55-00-45
122960 TLX

FRANCE

Data General Europe
61 Rue de Courcelles
4th Floor
75008 Paris, France
766-51-78
924-21-33
641227 TLX

Data General France
La Boursidiere, RN 186
Immeuble M BP 78
Route Nationale 186
92350-Le Plessis Robinson, France
(1) 630-21-05
200143 TLX

105-107 Rue De Crequi
69006 Lyon, France
(78) 5264-21
842-38-130 TLX

GERMANY

Data General, GMBH
Frankfurter Allee 27
6236 Eschborn
W. Germany
6196-48868
841-417434 TLX

Niederlassung West
403 Ratingen, Germany
Borsigstrasse 12
(2012) 46011/12
8-585-271 TLX

Niederlassung Hamburg
Kleine Bahnstrasse 10
200 Hamburg 54, West Germany
(40) 850-50-26
841-212448 TLX

Riesenfeldstrasse 75
8000-Muenchen 40, West Germany
089-3519011-4
05-24079 TLX

Talstrasse 172
D-7024 Filderstadt 1
711-702061

GREECE

Data General Middle East - Athens
260, Syngrou Avenue
Athens, Greece
958-0845
952-0557/956-0378
863-218189 DGME GR TLX

Corais S.A.
10 Stadiou Avenue
Athens 133, Greece
23-3714/7
863-215528 COR GR TLX

HONG KONG

Dataprep (HK), Ltd.
14th Floor, Block B
Watson's Estate
Hong Kong, B.C.C.
5-717-231
780-73184

INDIA

Ravi & Ashok Enterprises
S. No. 1-A, Irani Market Compound
Yervada
Poona, 411006
India
953-014-5275, 953-011-3552 TLX

Data Conversion Inc.
238 Main Street
Cambridge, MA. 02142
617-491-6827

IRAN

Compuran Corporation
Takht Tavous, Amir - Atabak St.
Varavinie St., No. 19
Tehran - Iran
83-9897/83-7286/5
215176 IRTA IR TLX
213182 IRSA IR TLX

ISRAEL

TEAM Computers and Systems, Ltd.
5 Tefutzot Israel Street
Givatayim, Israel
(03) 77-1031, -2, -3
341491 TLX

ITALY

Data General S.P.A.
Via Morigi, 3/A
20123 Milano, Italy
866-046
843-33243 TLX

Data General S.P.A.
Via Vincenzo Ussani 90
1-00151 Roma
(06) 5236691

JAPAN

Nippon Mini-Computer Corporation
Trade Dept.
Tohyo Bldg. 6-12-20
Jingumae
Shibuya, Tokyo
(03) 406-6451
781-242-5071 TLX

KOREA

M-C International (Korea) Ltd.
CPO Box 1355
Seoul, Korea
23-410/5
K24228 Emcee Korea Seoul

Room 1407
Center Bldg.
Seoul, Korea

717 Market Street
San Francisco, CA. 94103
415-543-1455

KUWAIT

Beidoun Trading Establishment
P.O. Box 3430
Kuwait, Arabian Gulf
42-0449, 43-1086, 41-0610
2368 BEIDOUN KT TLX

LEBANON

Data Systems S.A.R.L.
Corniche du Fleuve
Immeuble Steiger
P.O. Box 11-316
33-1336/32-5511/25-1310
20930 Riaudi Le TLX

LYBIA

Trading & Engineering Assoc., S.A.
Ben Ahour Street
P.O. Box 4494
Tripoly, Lybia
37283
Handiskawi Tripoly Lybia (Cable)

MALAYSIA

Dataprep (Malaysia) Sendirian Berhad
11th Floor, Angkasa Raya
Jalan Ampany
P.O. Box 2491
Kuala Lumpur
Malaysia
201898, 201696
784-30420 TLX

Sinar Sabah Sendirian Berhad
Room G-K, 5th Floor
Central Building
Kota Kinabalu, Sabah
East Malaysia
54671
SINARS MA80218 TLX

MEXICO

Control & Processo Electronico
Av. Juarez 14-20
Despachos 201-206
Mexico 1 D.F., Mexico
5-18-25-24, 5-12-85-19
1775826 CYPELME TLX

NETHERLANDS

Data General Holland, BV
Van Gijnstraat 17
Rijswijk Z.H.
The Netherlands
(70) 907-694
844-33545 TLX

PERU

Digita S.A.
Los Negocios 420
Surquillo Lima 34
Peru
45-5773, 47-1111, 41-1300
(36) 21083 PE DIGITA TLX

PHILIPPINES

Dataprep (Philippines), Inc.
G & S Building
250 Beundia Avenue
Makati, Metro Manila
Philippines
86-49-51
722-3968 TLX

PUERTO RICO

Pan American Computer Corporation
Banco Popular Center, Suite 1810
Hato Rey
Puerto Rico 00918
(809) 767-6505

SAUDI ARABIA

Rola Data Systems
P.O. Box 3172
Riyadh
Saudi Arabia
60-634
20172 NAJATI SJ TLX

SINGAPORE

Dataprep (S) PTE Ltd.
2nd Floor, Marina House
Shenton Way
Singapore 1
211311 DPREP
786-21249 TLX

SOUTH AFRICA

Perseus Computing & Automation, Ltd.
Dejong Centre, 2nd Floor
457 Rodericks Road
Lynnwood Pretoria
Republic of South Africa
47-27-82
960-30643, 960-30679 TLX

Perseus Computing & Automation, Ltd.
P.O. Box 4898
Johannesburg
Republic of South Africa
23-6158

SOUTH AMERICA

Data General Ltda.
Rua Georgia, 221
Caixa Postal 673
Sao Paulo, Sp. Brazil
543-0138, 241-0425, 542-8951
831-1122336 DGAL BR TLX

SPAIN

Madrid Hispano Electonica Informatica
Poligono Industrial de Urtinsa
Calle de las Fabricas
P.O. Box 48
Alcorcon (Madrid) Spain
6194108
831-42634 TLX

Figols 27-29
Barcelona - 14, Spain
3218595 & 3219085

Zabalbide, 42
Bilbao-6, Spain
4238309 & 4238062

Navarro Reverter, 2
Valencia-4, Spain
3731297

Sanchoel Sabio, 28
San Sebastion, Spain
462554

SWEDEN

Data General Sweden AB
Industrivagen 20
S-17148 Solna, Stockholm
Sweden
8083-4400
854-10089 TLX

Doktor Forselius Gata 1
S-42326 Gothenburg
Sweden
31-81-31-90

Carlsgatan 54
S-211 20 Malmoe
Sweden
30-10-35-85

SWITZERLAND

Data General A.G.
Badenerstrasse 567
8048 - Zurich, Switzerland
1-54-17-82
845-58437 TLX

Data General, S.A.
Avenue de Valmont 12
CH-1010 Lausanne, Switzerland
021-33-33-34 or 35
845-26105 TLX

TAIWAN

Dataprep (Taiwan)
P.O. Box 53-316
Taipei, Taiwan
771-0515
785-23838 TLX

UNITED KINGDOM

Data General, Ltd.
London Office
Westway House
320 Ruislip Road East
Greenford
Middlesex, England
1-578-9231
851-935364 TLX

Manchester Office
Nelson House, Park Road
Timperley, NR Altrincham
Cheshire, England
(61) 969-3935/9
851-667903 TLX

Swan Office Centre
1506/8 Coventry Road
Yardley
Birmingham, England B258AD
021-707-3433
851-339392 TLX

SCOTLAND

Data General, Ltd.
2nd Floor
Hellenic House
87-97 Bath Street
Glasgow G2 2EE
Scotland, UK
41-332-3205
851-779258 TLX

URUGUAY

Lumiere S.A.
Rincon 531-ESC. 401
Montevideo, Uruguay
90-4774
(32) 546 AB QUESMAR UR TLX

U.S.S.R.

Codevintec Pacific Inc.
6263 Variel Drive
Woodland Hills, CA 91364
213-348-1719
698474 TLX

VENEZUELA

Codata
Avda 2 No. 86A-27
Apartado 776
Maracaibo, Venezuela
22-3251, 22-3966, 22-3671
(DDI, Dial 0058-61 + phone no.)
31-61140 SAVEN MEX TLX

OEM's

MEXICO

Maquinas de Informacion S.A.
Insegentes SUV No.1722-202
Mexico 20 D.F.
524-9195

NICARAGUA

Analisis Y Sistemas
Belmonte # 31
Apartado # P-234
Managua, Nicaragua
50235

PORTUGAL

Equipamentos de Laboratorio, Ltd.
P.O. Box 1100
Lisbon, Portugal
976551
832-12702 TLX

SRI LANKA (CEYLON)

Mahiaraja Organization
7 Braybrooke Place
Colombo 2
Sri Lanka
Cable 1195 Colombo Rendezvous

TURKEY

Sisag Company Ltd.
CINNAH CADDEST 5/4 Kavaklidere
Ankara, Turkey
27-7470 (direct)
26-4729, 27-8105, 27-8209
42747 SSAG TR TLX

MANUFACTURING SUBSIDIARIES

HONG KONG

Data General Hong Kong, Ltd.
Kaiser Estates, 5th Floor
Man Yue Street
Hung Hom
Kowloon
Hong Kong
K-634381-4

THAILAND

Data General Thailand, Ltd.
2189 New Petchburi Road
Bangkok
Thailand
916357

READERS COMMENT FORM

DOCUMENT TITLE: .....

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

How did you use this publication?

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

Did you find the material:

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

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

COMMENTS

PAGE	COL	PARA	LINE	FROM	TO

From:

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

Data General Corporation
ENGINEERING PUBLICATIONS
COMMENT FORM
DG-00935

CUT ALONG DOTTED LINE

FOLD DOWN FIRST FOLD DOWN

FIRST CLASS
PERMIT NO. 26
SOUTHBORO
MASS. 01772

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

Postage will be paid by:

DataGeneral
Southboro, Massachusetts 01772

ATTENTION: Engineering Publications

FOLD UP SECOND FOLD UP

Data General

Data General Corporation, Westboro, Massachusetts 01581