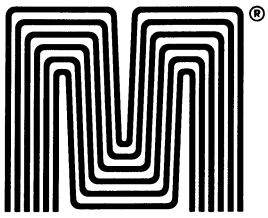
32/S COMPUTER REFERENCE MANUAL

$10.00

32/S COMPUTER REFERENCE MANUAL

RM 20003250
&
RM 20003250-1

JANUARY 1975



Printed in U.S.A.
98800 76 1018A

Microdata Corporation
17481 Red Hill Avenue, Irvine, California 92714
Post Office Box 19501, Irvine, California 92713
Telephone: 714/540-6730 · TWX: 910-595-1764

FOREWORD

This manual describes the 32/S1, an extended capability version of the basic 32/S computer. All programs that run on the 32/S, and I/O controllers which interface to the 32/S, will also run on the 32/S1.

The 32/S1 features that are not provided in the 32/S are:

1. three of the doubleword arithmetic instructions:

   DMUL - Doubleword Integer Multiply
   DDIV - Doubleword Integer Divide
   DMOD - Doubleword Modulo

2. floating point instructions, including arithmetic, and miscellaneous utility instructions:

   FADD - Floating Point Add
   FSUB - Floating Point Subtract
   FMUL - Floating Point Multiply
   FDIV - Floating Point Divide
   FABS - Floating Point Absolute Value

   FEQ - Floating Point Equal Comparison
   FGE - Floating Point Greater Than or Equal Comparison
   FGT - Floating Point Greater Than Comparison
   FLE - Floating Point Less Than or Equal Comparison
   FLT - Floating Point Less Than Comparison
   FNE - Floating Point Not Equal Comparison

   STT - Store Tripleword
   LTW - Load Tripleword
   LTL - Load Tripleword Literal

   FLOT - Float an Integer
   FIX - Fix a Floating Point Number

   FMAX - Floating Point Maximum Value
   FMIN - Floating Point Minimum Value
   FSGN - Floating Point Sign Value

   (L field value of 3 in Load Address instruction which specifies a tripleword index.)

3. string manipulation instructions:

   MOV - Move String Within Stack
   MVP - Move String from Procedure to Stack
   MVA - Move String Absolute

   SLT - String Compare Less Than
   SLE - String Compare Less Than or Equal
   SEQ - String Compare Equal
   SNE - String Compare Not Equal
   SGE - String Compare Greater Than or Equal
   SGT - String Compare Greater Than

4. maximum, minimum, and sign value instructions:

   MAX - Maximum Value
   MIN - Minimum Value
   DMAX - Doubleword Maximum Value
   DMIN - Doubleword Minimum Value
   SGN - Sign Value
   DSGN - Doubleword Sign Value

5. memory reference swap instruction:

   SWAP - Swap Word in Stack with Memory

6. internal interrupt numbers:

   9, argument 1 - Stack Overflow
   9, argument 2 - Stack Underflow
   9, argument 4 - SB/SL Violation
   9, argument 7 - Store Into Program Segment Violation

Additionally, the bits of the external interrupt mask in the Program Status Register are redefined as follows:

Bit Position	32/S	32/S1
4	{ operator interrupt external interrupt line 0 }	operator interrupt
5	{ timer interrupt external interrupt line 1 }	timer interrupt
6	external interrupt line 2	external interrupt line 0
7	external interrupt line 3	external interrupt line 1

These same redefinitions apply to the corresponding bit positions of the external interrupt mask within the Interrupt Vector Table entries and within Marks. Note that external interrupt lines 2 and 3 still exist in the 32/S1, but cannot be disabled.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS - (Continued)

1 Introduction

1.1 THE 32/S COMPUTER, 3200 MICROPROCESSOR, & MPL

The Microdata 32/S Computer is a new concept in general purpose minicomputer design. The 32/S is microprogrammed on the state-of-the-art computer hardware of the Microdata 3200 Microprocessor. Its unique architecture makes it feasible to reduce programming costs by doing all programming in a high-level language, the system-oriented Microdata Programming Language (MPL).

The Microdata 32/S is a push-down stack architecture computer implemented via firmware on the microprogrammable Microdata 3200. Its architecture was designed in conjunction with the design of the Microdata Programming Language, MPL, a high-level programming language. MPL provides the user with all of the time and cost-saving benefits of high-level language programming, while providing all the capabilities required by systems programmers. The code produced by the MPL compiler for the 32/S is as efficient as the machine code which can be obtained with assembly language programming on a conventional architecture computer.

The 3200 Microprocessor

The 3200 Microprocessor is a 16-bit machine with 4K to 128K words of 300 nanosecond MOS main memory, addressable to the byte level. It is microprogrammed with a bipolar 32-bit control memory which is expandable to 4K words, and which operates with a 135 nanosecond cycle time. Software level (32/S) instruction look-ahead logic and top-of-stack registers are provided in hardware to maximize speed.

The 3200 utilizes a common bus, the Monobus, for accessing all main memory modules and I/O device controllers. Memories and controllers of various speeds may be mixed on the asynchronous Monobus and uniformly accessed with standard memory reference instructions. Overlapped bus requesting and data transferring permits very high-speed data transfers.

Input/output can be byte or word oriented under program control, or block-oriented under either computer control ("concurrent I/O") or controller hardware control (direct memory access). Four external interrupt lines establish the relative priorities of groups of I/O device controllers. Relative priority among the controllers on each line is established by their positions along the Monobus. Each I/O device controller may be manually assigned to a specific address and interrupt line. A unique interrupt processing procedure and environment may be specified for each I/O device address.

The 32/S Computer

The 32/S computer is organized around an operational data push-down stack. The last-in/first-out properties of the stack eliminate the necessity to allocate processing registers, store and retrieve temporary data, or assign memory locations for temporary data and parameters.

The 32/S instructions were designed with the requirements in mind of the MPL compiler (and other compilers). Special instructions are included to operate on data in the stack, establish and switch operating environments, switch between multiple user stacks, and handle a variety of data types. In addition, user specified instructions can be implemented in the 3200 firmware and accessed through MPL without the need to modify the compiler.

The 32/S main memory is allocated into separate data stack areas for each user and one or more program segment areas. All memory references are specified relative to individual stack and program base registers; therefore, both stacks and program segments may be loaded and dynamically relocated within available memory. A reference to a program segment absent from main memory can be made to cause an executive interrupt for the purpose of loading the missing segment.

The maintenance front panel of the 32/S is specially designed to permit the user/programmer to monitor the operation of the computer, to interrogate various software level registers and relocatable memory locations, or to inspect actual microlevel registers and buses. Switches are provided for breakpoint address, control memory instruction, and selection of address and data displays. A keyswitch provides the basic run/stop/off and panel lockout control. It also provides a standby power position to maintain information in the MOS main memory. A manual interrupt button and an initial program load button are also provided. A simpler, less-expensive, basic front panel, which provides the keyswitch, load, and interrupt buttons is provided for actual system installations.

MPL

MPL is a block structured high-level language based upon PL/I. MPL programs are written in the form of statements which are organized into PROCEDURE blocks and BEGIN blocks. The types of statements include: assignment, CALL, DO, DECLARE, END, GO TO, IF-THEN-ELSE, BEGIN, PROCEDURE. REPEAT and RETURN. The blocks of statements may either be separate or nested one within another. Data is declared and described with attributes within the block of statements which references it. Such data declared within a block is available to all blocks nested within that block. Depending upon where within nested blocks the data is declared, the storage space for this data may be assigned either statically or dynamically. The data may be either scalar or arrays.

I/O is handled in MPL by assigning variable labels to the registers of the I/O device controller on the Monobus, and then accessing these registers by name, in the same way as any other variable. Facilities are provided to enable the programmer to write interrupt routines to service the interrupts generated by I/O device controllers.

Note that, although MPL is a high-level language which looks very similar to other high-level languages (such as PL/I and Fortran), MPL is not a machine independent language. It is very specifically a system-oriented language for the 32/S computer which serves as a replacement for the conventional lower level assembly language used on other minicomputers.

The relationship of the 3200 microprocessor, the 32/S computer, and the MPL machine are summarized in Figure A. This manual presents the 32/S computer. The Microdata 3200 Reference Manual and the Microdata Programming Language Reference Manual are available as separate documents.

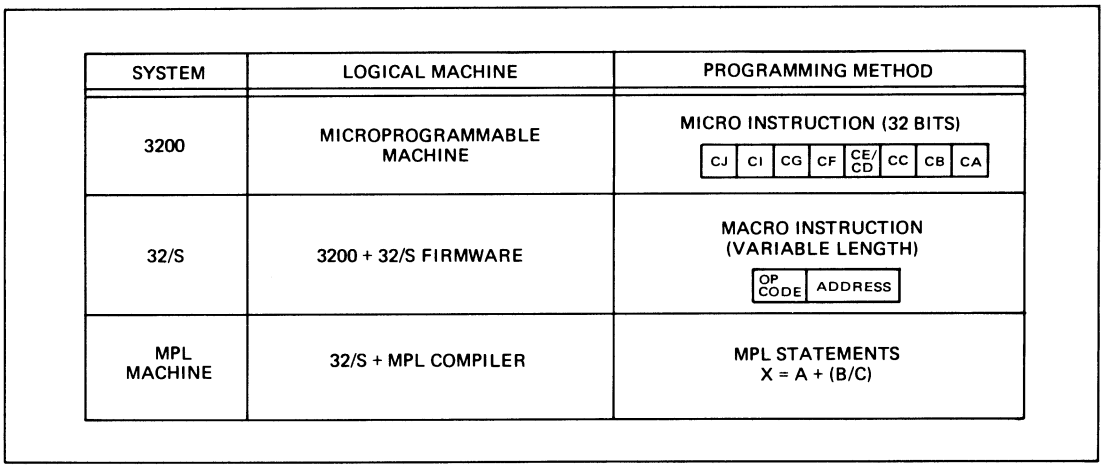


Figure A. The 3200, 32/S, MPL Heirarchy.

MOS Memory

- Expandable to 256K bytes (128K words)
- 8K and 16K byte (4K and 8K word) plug-in modules.
- Parity (1 bit per byte) option.
- Speed:

  350 nsec access time.
  450 nsec read cycle time
  350 nsec write cycle time

Real-Time Clock

- AC line frequency (100/120 Hz).
- Custom frequency crystal-controlled option.

Initial Program Load:

- Initiated with front panel button.
- Load from any specified device.

Power Fail Detect and Automatic Restart

Speed:

- 135 nanosecond clock.
- Instruction execution time variable from 400 nanoseconds (depending upon instruction prefetch, push-down stack state, and instruction type).
- Maximum DMA-memory transfer rate of 500 nanoseconds per 16-bit word.
- CPU-memory transfer rate of 800 nanoseconds per 16-bit word.
- Monobus control access time dependent upon priority of Monobus requestor, whether a Monobus operation is in progress, whether a request for a next Monobus operation has been granted, and whether a memory refresh cycle is outstanding. (See I/O Interface Manual.)

Figure A. 32/S Computer, Table-Top Version.

- Automatic transfer from AC power to battery pack (when pack is available) upon loss of AC line. Battery pack maintains data in MOS memory while CPU power-fails down. CPU operation is reinitiated and battery is switched off when AC power is resupplied.
- Provision made in design of backplane to supply additional +5V current from a secondary rack mounted power supply in systems that require more than 35 Amps. When this option is installed, the last seven slots, not including the power supply slot at the rear, are powered from the remote power supply. Both power supplies are slaved to operate together upon loss of AC line and return of AC line.

Environment:

- 0°C to 50°C.

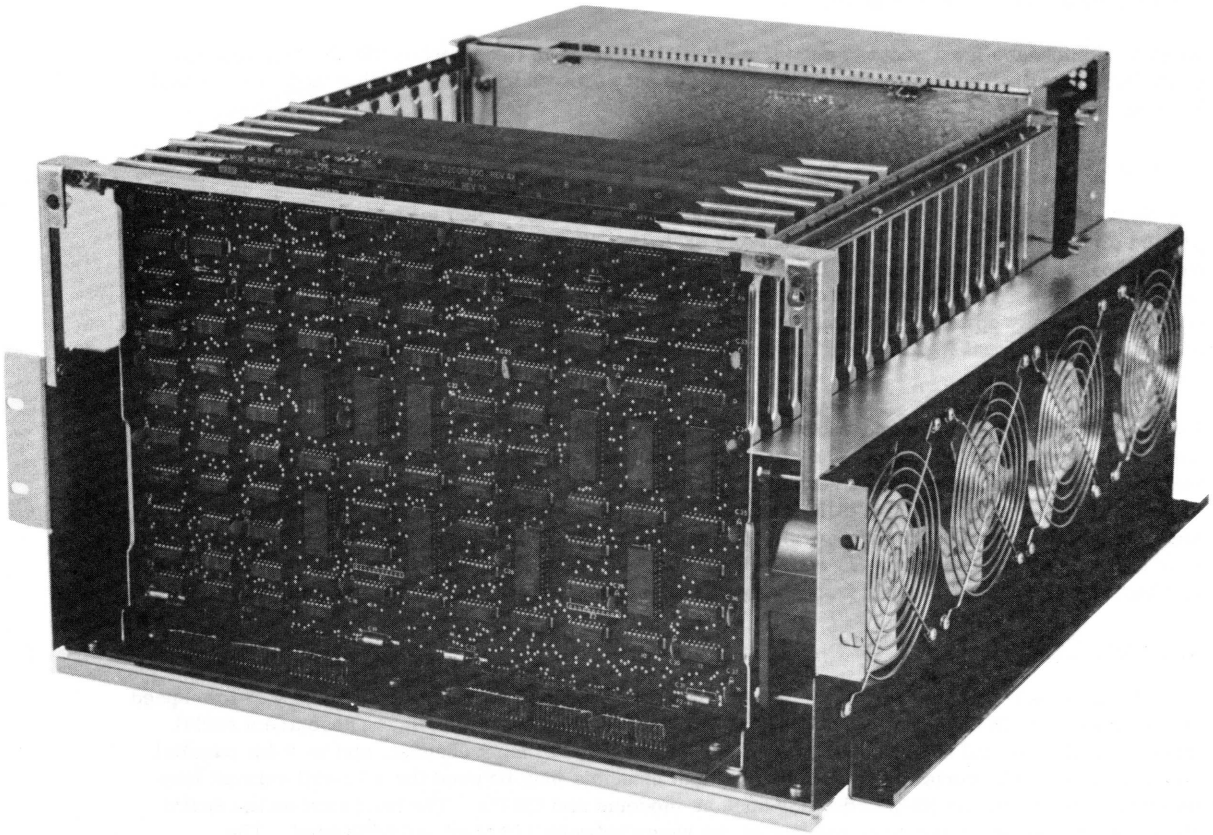


Figure A. 32/S Computer, Cover Off.

1.2 32/S SPECIFICATIONS - GENERAL

The Microdata 32/S is a 16-bit, 135 nanosecond clock, 350-450 nanosecond MOS memory cycle, push-down stack architecture computer. It provides an addressing range of up to 256K bytes on a common I/O-memory Monobus.

Push-Down Stack Architecture:

- Push-down stack replaces conventional multiple accumulator/index registers.
- Hardware registers provided for top five levels of stack.
- Stack continued in main memory to a maximum depth of 32K words.
- Specialized instructions provided to create mark entries within the stack to delineate PROCEDURE and BEGIN block working areas within the stack and to roll back these working areas or environments when blocks are exited.
- Instructions which reference memory are provided with addressing modes which correlate with data defined locally within a block's environment and data located within the environment of a sequence of nested blocks.

Monobus Architecture:

- Common Monobus interface to both memory and I/O device controllers.
- Asynchronous control of Monobus, permitting interface to different speed memories and controllers.
- 16-bit data.
- 18-bit address (addressing to byte level).
- Overlapped Monobus request-for-control and transfer operations.

Data:

- 16-bit paths within CPU and on Monobus
- Specialized instructions available to access/store byte, word, doubleword, tripleword and variable bit fields within a word.
- 16-bit and 32-bit binary integer arithmetic and floating point arithmetic.

Memory Referencing:

- Program segments relocatable (all program references relative to hardware Program Base Register).
- Data stack areas relocatable (all data stack references relative to hardware Stack Base Register).
- Virtual memory (any call to new program segment references a Program Library for the location of the segment; an internal interrupt will be caused if the Program Library entry is flagged).

Instruction Set:

- Variable length instruction format.
- Automatic 16-bit instruction prefetch logic.
- 151 instruction types:
  - 15 which transfer data to/from the top of the stack.
  - 88 which operate on data in the top of the stack.
  - 9 which manipulate character strings.
  - 17 branch.
  - 22 control.

I/O Architecture:

- Software programmed transfers.
- Concurrent block transfers controlled by CPU hardware.
- Direct memory access (DMA) block transfers controlled by I/O controller hardware.

Interrupt Architecture:

- Enable/disable within I/O device controllers.
- Four external interrupt lines
  - Two maskable in Program Status Register.
  - Daisy-chain priority along each line.
- Unique interrupt vector for each type of internal interrupt and for each I/O device controller.
- Arm/disarm bit per interrupt vector.

1.3 32/S SPECIFICATIONS - MECHANICAL & POWER

The 32/S chassis provides 15 card slots, an integral power supply, and integral cooling. In its rack mount version it is 21 inches deep and 10.5 inches high. An optional battery pack maintains data storage in the MOS memory in the event of power failure.

Front Panel:

- Maintenance front panel
- Basic front panel

Chassis Dimensions:

- Desk top configuration: 10.5 inches high; 19 inches wide; 22.75 inches deep
- Rack mount configuration: 10.5 inches high; 17.75 inches wide; 21 inches deep (from rack mounting rails)

Mechanical Design:

- Monobus and other processor buses provided in multilayer backplane along bottom of chassis
- 15 card slots provided (plus specialized card slot for power supply)
  - 3 card slots for the processor (including 32/S microprogram control memory)
  - 1 card slot for maintenance front panel logic board (if maintenance front panel is used)
  - 1 card slot for each 16K byte MOS memory module
  - 1 card slot for each I/O device controller (except disc controller which requires 2 slots)
- The front panel may be removed from the front of the enclosure to gain access to the front card slot. Any card may be placed in this slot, thus eliminating the necessity for card extenders when performing maintenance.
- 4 high-power cooling fans provide cross-ventilation for cards and power supply at all times.
- Channel for I/O device cables available along the upper half of each side of cards.
- Card size is 9" x 14".
- Optional expansion chassis available for additional MOS memory modules and/or I/O controllers.

Power:

- 115/230 VAC ± 10%, less than 500W for average configuration.
- Standard integral power supply provides the following power:
  - 35 Amps at +5V
  - 6 Amps at +5.3V (used in MOS memory modules)
  - 3 Amps at -12V
  - 3 Amps at +12V
  - 4 Amps at +21V (used in MOS memory modules)
- Optional Gel Cel battery pack available to supply power to maintain information in MOS memory.

1.4 SYSTEM HARDWARE

The 32/S computer is implemented with standard and optional printed circuit board modules which may be plugged into any one of 15 available slots in a printed circuit board backplane.

The modular basis of the 32/S computer is the 9" x 14" printed circuit board module. A 14" edge plugs into a pair of 50-pin connectors in a multi-layer printed circuit board backplane. For I/O device controller modules, connectors for I/O cables are attached to the upper half of either of the 9" sides. The printed circuit board backplane contains the wiring for the Monobus, for control memory buses, and for CPU buses. Every signal is available at the same pin position of each of the 15 available card slots within the chassis. This permits any module to be placed in any position within the chassis. Since the front panel can be snapped off and hung to one side, leaving the printed circuit module in the first slot accessible from the front, this means that any printed circuit module can be serviced without the use of an extender board by simply plugging it into the first slot.

CPU and Front Panel

The 32/S CPU itself, with the 32/S firmware, is contained on three printed circuit board modules: the Monobus interface board, the data board, and the processor control board. The read-only memory IC's for the firmware are contained on the processor control board. These boards are discussed in the Microdata 3200 Reference Manual.

When a maintenance front panel is used it must be interfaced to the Monobus via the maintenance panel logic board. The maintenance logic board is not required when the simpler basic front panel is used.

Memory

The MOS main memory is implemented in 8K by 16-bit modules. Each module is a standard size printed circuit board assembly which contains 8K bytes (4K words) of memory on the main printed circuit board, and an additional 8K bytes of memory on a piggyback printed circuit board which is bolted down to the main board. An optional 8K byte module is provided by not attaching the piggyback board. The parity option is implemented with an additional bit per byte of storage in the module; corresponding optional logic is provided in the CPU. Up to 16 modules may be logically placed on the 32/S Monobus.

Each memory module contains a 4-bit selector switch to permit it to be manually set to respond to desired 16K-byte address bank. The memory module contains its own control logic, including that required for automatic refresh in response from signals supplied by the power supply. A block of 32 addresses in each of the 128 1K MOS memory IC's are refreshed simultaneously every 30 microseconds during normal computer operation. This operation takes 300 nanoseconds and has priority over data transfer operations. To reduce battery drain during standby, all addresses are refreshed in a burst at an interval ranging from every 2 milliseconds at 25°C to every 1 millisecond at 50° C.

I/O Device Controllers

A multi-purpose I/O controller, the MPI/O controller, provides interfaces to a variety of low-speed I/O devices. The MPI/O controller contains four device interface channels. These are a serial input channel, a serial output channel, an 8- or 12-bit parallel input channel, and an 8-bit parallel output channel. The serial input and serial output channels may be used for a 20-mil current loop teletype interface or for RS-232 devices such as modems and CRT's. The baud rate on the serial channels may be set to any standard rate in the range between 110 baud and 9600 baud. The

parallel input channel may be used with a punched card reader or a paper tape reader. The parallel output channel may be used with a line printer or with a paper tape punch. Both serial and parallel channels communicate with the 32/S via both programmed I/O and concurrent I/O.

The disc controller is a two-board device controller option. It provides an interface to up to four Microdata disc files. These disc files contain one fixed disc and one removable cartridge and may be either 5 megabyte or 10 megabyte units.

The magnetic tape controller option provides an interface to up to two industry-standard magnetic tape formatters. Each of these formatters can control up to four magnetic tape units. These tape units may be various mixtures of NRZI and phase-encoded and may operate at speeds of 12.5 to 120 inches per second with densities ranging from 200 to 1600 characters per inch.

The communications controllers provide multi-channel interfaces for TTY, CRT, and modems. One model of asynchronous communications controller provides parallel modem control lines along with the serial data line for each channel. Baud rates are programmable in this unit. The other asynchronous communications controller is a simpler unit which provides an interface only to current loop or RS-232 devices. A third communications controller provides a bisync interface to synchronous modems.

Additional custom design controllers can be supplied either by Microdata or implemented by users. Design information is provided in the Microdata 3200 Interface Manual.

Control Memory

The control memory contained in the processor control board can be expanded with an additional read-only control memory expansion board. This board can be used to expand the 32/S firmware to a total read-only memory capacity of up to 4K words. For developmental purposes, writable control memory boards are available. These provide the control memory function in bipolar read/write memory IC's and can be loaded by the 32/S computer from main memory or from an I/O device and then used to control the 32/S computer.

Expansion Chassis

A bus coupler board is used to bring the Monobus out of the chassis to an optional expansion chassis. Memory and I/O device controllers may be placed in the expansion chassis.

Figure A indicates the type of printed circuit modules which are available as both standard and optional items. It also indicates the interconnection, within the printed circuit board backplane, between these models.

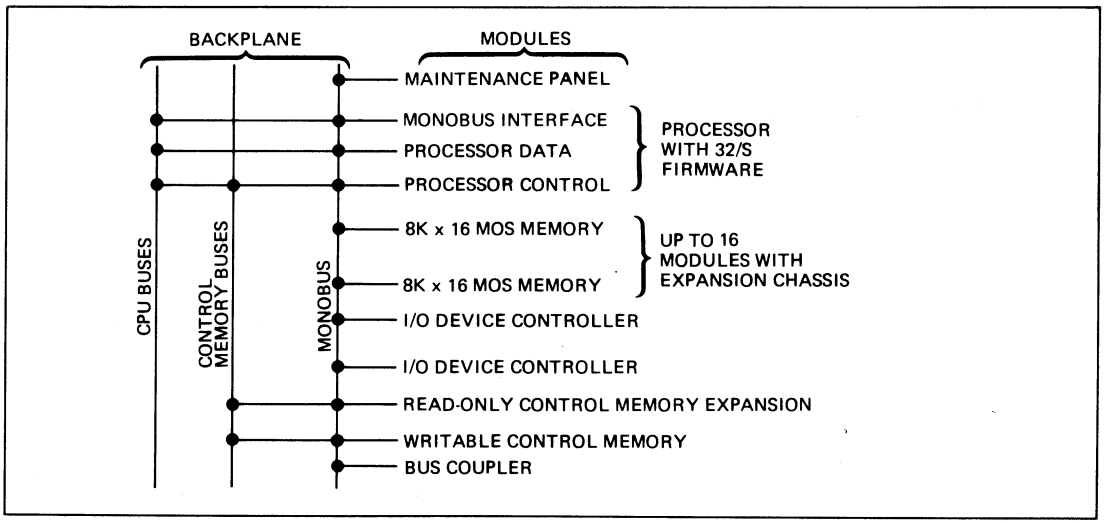


Figure A. 32/S System.

1.5 THE PUSH-DOWN STACK CONCEPT

The compilation of high-level languages, particularly block-structured languages with expressions, is most easily performed for a machine with a push-down stack architecture.

The reason the 32/S can be efficiently programmed in a high-level block-structured language such as MPL is that translation to machine language is simpler for machines with a "push-down stack" architecture than for machines with fixed operational registers. This architectural concept has been successfully proven in large-scale processors. And since non-block structured languages, such as FORTRAN, are degenerate examples of block structured languages the availability of a minicomputer with a stack architecture makes it easier to obtain a compiler for most high-level languages.

It is difficult for a compiler to generate efficient programs for a conventional multiregister architecture. The human programmer, shuttling data from register to register in a minicomputer of conventional design, can visualize what is going on with more perception than can a compiler. Therefore the compiler is likely to use the registers in a very constrained way, with many needless transfers to and from the main memory. The push-down stack architecture, on the other hand, provides the facilities which a compiler can use extremely efficiently.

Conceptually, a push-down stack is a set of memory locations, of which only the most recently used one is accessible. Data is loaded or "pushed" into the stack through this one register; when additional data is loaded, previously loaded data moves sequentially "down" from one register to the next in the set (Figure A). Data is retrieved or "popped" from the same register used for loading; each such retrieval causes other data to move "up" sequentially from other registers. All data is retrieved in the reverse order from that in which it was loaded, so that the push-down stack is sometimes called a "last-in-first-out" (LIFO) buffer.

No actual push-down stack works like the conceptual model described above. Instead, in the simplest configuration, a set of successive locations in the computer's main memory is set aside for use as a stack. Data does not actually move from location to location during a push or a pop. Instead, the address of the location which is currently the top of the stack is stored in a stack pointer which can be incremented or decremented by 1 to effect the push and the pop. (Figure B).

To obtain a significantly faster operating stack, the 32/S uses five high-speed hardware registers with a separate pointer for these stack head registers, and a stack pointer and main memory locations for the lower portion. The highest register is tied directly to one input of the computer's arithmetic unit. Since most of the traffic in and out of the stack involves only the top few locations, the hardware registers reduce the number of accesses to the main memory and speed up the system's operation.

In a push-down stack architecture, the instructions that concern the stack automatically transfer data as needed between the hardware register portion of the stack and its extension in the main memory, so that this division is transparent to the user. Thus, a push into the register stack when it is already full is automatically preceded by a transfer of the word in the bottom hardware register into the main memory. But, for maximum performance, the reverse is not necessarily true. A pop from the stack is not accompanied by a transfer from the main memory stack; the pointer to the stack head registers is decremented if the top of the stack is in these registers, and the stack pointer (to main memory) is decremented if these registers are all empty. Data is moved from main memory to the stack head registers only if a stack operation (e.g., ADD) requires operands which are in the main memory.

The simplest example of stack usage is the evaluation of an arithmetic expression. Working in a high-level language, a programmer writes an expression such as:

(A - B)/ (C+D)

The compiler translates this into the format known as "reverse Polish notation," after Jan Lukasiewicz, a Polish logician:

AB-CD+/

This notation eliminates parentheses because each mathematical operator refers not to the two operands between which it stands, but to the two operands preceding it. Carrying out the computation after translation is simple because the computer can perform each mathematical operation as it encounters the operators, without backing up to see what went before, or waiting to see what comes next.

To evaluate an expression in Polish notation, a compiler produces a LOAD instruction (a "push") for each variable and a mathematical instruction (such as ADD) for each operator in the expression. Mathematical instructions operate on the top one or two items in the stack and replace them with the result. This sequence is illustrated in Figure C.

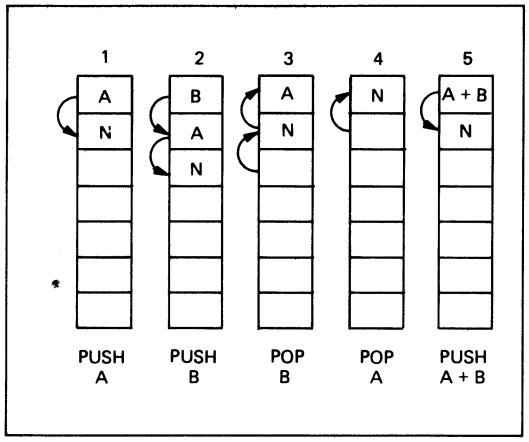


Figure A. Push-Down Stack Concept.

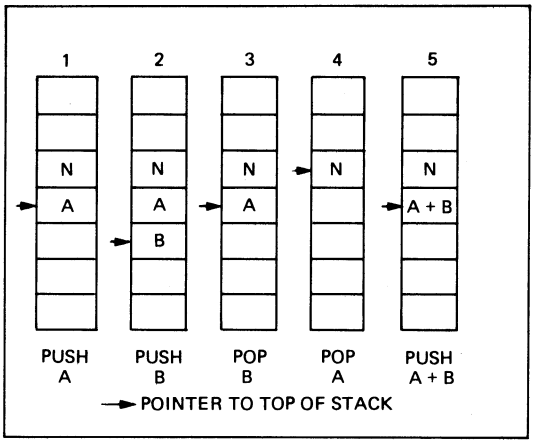


Figure B. Push-Down Stack Implementation in a Memory.

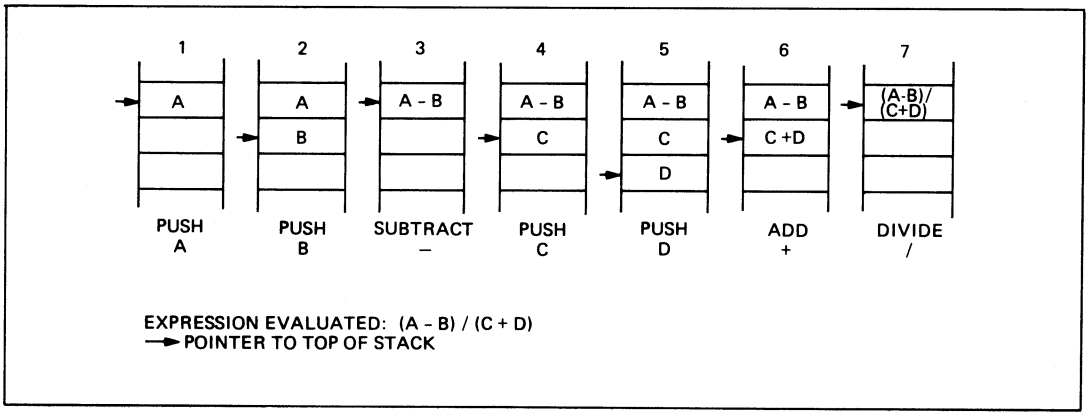


Figure C. Expression Evaluation in a Push-Down Stack.

1.6 CONVENTIONS USED IN THIS MANUAL

This manual is written in a modular format with each pair of facing pages presenting a single topic. Ideas are explained as much as possible with easy to visualize and easy to recall drawings.

The approach taken in this manual differs greatly from the typical technical manual. Here each pair of facing pages discusses an individual topic. Generally the left hand page is devoted to text and the right hand page to figures referred to by that text. At the head of the text page there are a pair of titles, the first one being the section and the second one being the topic. Immediately below these titles is a brief summary of the material covered in that topic. The advantage of this format will become readily apparent to the reader as he begins to use the manual. First of all, the figures referred to in the text are always conveniently right in front of the reader at the point where the reference is made. Secondly, there is a psychological advantage to the reader in knowing that, when he has completed reading a topic and goes to turn the page, he is done with one idea and ready to encounter a new idea.

The figures used in this manual are mostly drawings of Monobus maps with the contents of Monobus locations being shown and the addresses of particular Monobus locations being indicated. The Monobus locations include main memory and I/O device controller registers. Certain conventions are followed throughout this manual to simplify these drawings and thereby enable the reader to concentrate on the concepts rather than upon unnecessary detail.

The Monobus map is always shown as a rectangle which indicates a range of Monobus locations. (See Figure A). In these drawings Monobus address always increases as one moves downward on the page. The Monobus map is shown as a succession of 16-bit word locations. Often the individual word locations are drawn in when this is needed to explain an idea.

The 32/S Monobus provides a 16-bit data path and 18 address lines. The data transferred, however, may be specified (by control lines) to be either a 16-bit word or an 8-bit byte. In a byte transfer the least significant address bit specifies the least significant byte (bit = 1) or the most significant byte (bit = 0) of the 16-bit data path. In a word transfer the least significant address bit is ignored by the hardware. However, by convention, the 32/S firmware uses a least significant address bit of zero when executing a word transfer. Addresses of Monobus word locations and of most significant byte locations are therefore both even number, while those for least significant bytes are odd.

Most of the addresses shown in the figures of this manual are word level addresses. For example, the SB register is an 18-bit register which contains the base of the current pushdown stack in main memory. (The contents of the SB register are always a multiple of 4 because, by definition, the stack is a succession of word locations beginning at an even word address.)

Addresses are shown for Monobus locations in the manner illustrated in Figure B. The (1) portion of this figure illustrates a detailed representation. SB is shown pointing at a word location on the Monobus. This indicates that the contents of the SB register contain the address of that word location. The other Monobus location is shown relative to the location pointed to by SB. The relative displacement of the second location from the first one is contained in the EP register.

To simplify the drawing, the actual registers are not shown. Instead their names are used with arrows as shown in the (2) portion of Figure B.

In order to further simplify the drawings, as the reader becomes more familiar with the definitions of the main registers, the arrows used in pointing at Monobus locations and in showing relative displacements are replaced by simply putting the register name with a colon in front of the Monobus location. This is shown in the (3) portion of Figure B.

Finally, hexadecimal numbers are differentiated from decimal numbers by enclosing them in quotation marks, e.g., "0A51", is a hexadecimal number.

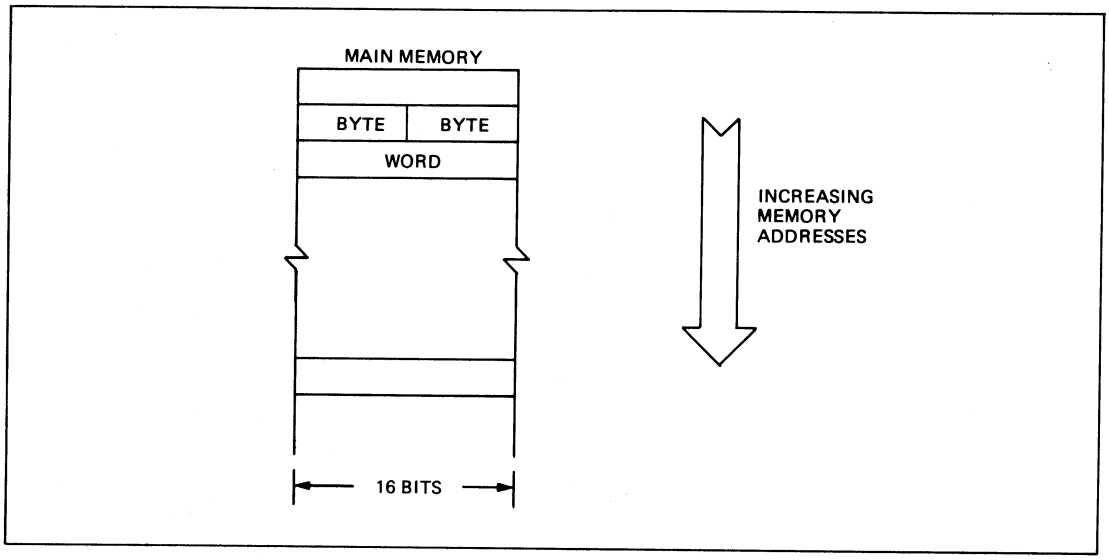


Figure A. Memory Drawing Conventions.

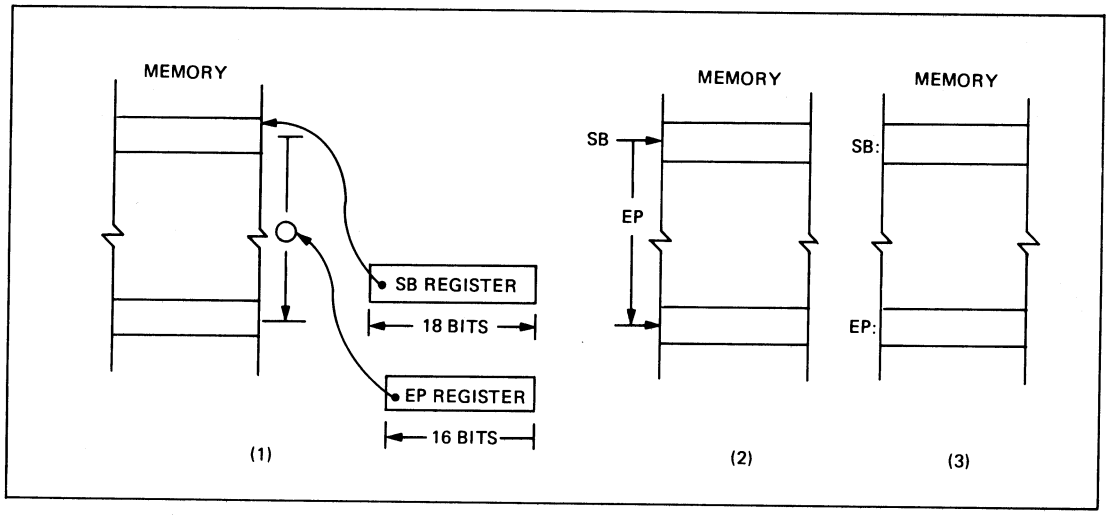


Figure B. Memory Address Drawing Conventions.

2 Organization

2.1 MONOBUS ORGANIZATION

The Monobus provides an addressing range of 256K bytes. Main memory is divided into program segments and data stacks, with active program segment and data stack areas specified by a set of seven 18-bit and 16-bit registers. I/O device controllers and control memory are assigned to Monobus addresses in ranges above those used by main memory.

The Monobus has an addressing range of 256K bytes. Modules on the Monobus include main memory, I/O device controllers, and control memory. See Figure A.

The Monobus addressing range is divided into four 64K-byte banks. This division into banks results from the fact that all address arithmetic is performed on the least-significant 16 bits of the 18-bit byte-level Monobus addresses. No carry from this 16-bit arithmetic is propogated into the most significant 2-bit field of the Monobus addresses. Therefore, addresses which should cross into the next bank when incremented or when increased by a displacement or index will instead wraparound to the beginning of the same bank.

Main memory is provided in 16K and 8K byte modules. Each module has a 4-bit switch to select the 16K range of Monobus addresses for that module. To obtain a contiguous memory, modules would be assigned sequential 16K ranges starting at Monobus address 0. Because of the relocatable features of the 32/S, it is possible to run programs with a non-contiguous address space.

The control memory interfaces with the CPU for control purposes via the control memory bus. However, in addition, control memory provided on an optional read-only control memory board can be read via the Monobus, and control memory provided in optional writable control memory modules can be read or written via the Monobus.

Control memory is addressable on the Monobus starting at byte location "38000" (224K). Addresses correlate with addresses of the control memory on the control memory address bus. (See topic 10.1).

I/O device controllers are assigned eight-word blocks of Monobus addresses, referred to as "Device Register Blocks." A multi-channel controller, such as the MPI/O, has a Device Register Block for each device it controls.

Standard Microdata controllers are wired for Monobus addresses within the block starting at byte location "3C000" (240K). A switch is provided on each controller to select one of 1024 "device numbers" for each device register block associated with the controller. (See topic 4.2.)

Within main memory, the first 16 word locations are reserved. (See topic 2.15). In addition, up to 512 words beginning at location 16 are reserved for the program library, PLIB, a table of pointers to program segments. (See topic 2.3).

The remainder of main memory, above PLIB, is assigned, by the loader program and/or the software operating system, to program segments and data stacks. Program segments and data stacks may be of any length and may be positioned within memory in any sequence. However, each program segment and each data stack must be wholly contained within a 64K byte bank. At any time one program segment and one data stack are active.

Three registers define the active program segment: the Program Base, PB, specifies the base address of the segment; the Program Pointer, PP, specifies the address of the 32/S instruction to be executed, relative to PB; the Program Length, PL, specifies the size of the segment. The program segment is discussed in the next topic.

Four registers define the active data stack: the Stack Base, SB, specifies the base address of the stack; the Environmental Pointer, EP, specifies the location, relative to SB, of the "Mark" entry in the stack which identifies the beginning of the current "environment" in the stack; the Stack Pointer, SP, specifies the location, relative to SB, of the top of the stack in main memory. The Stack Length, SL, specifies the maximum size allocated to the stack. The data stack is discussed in a sequence of topics starting with topic 2.5.

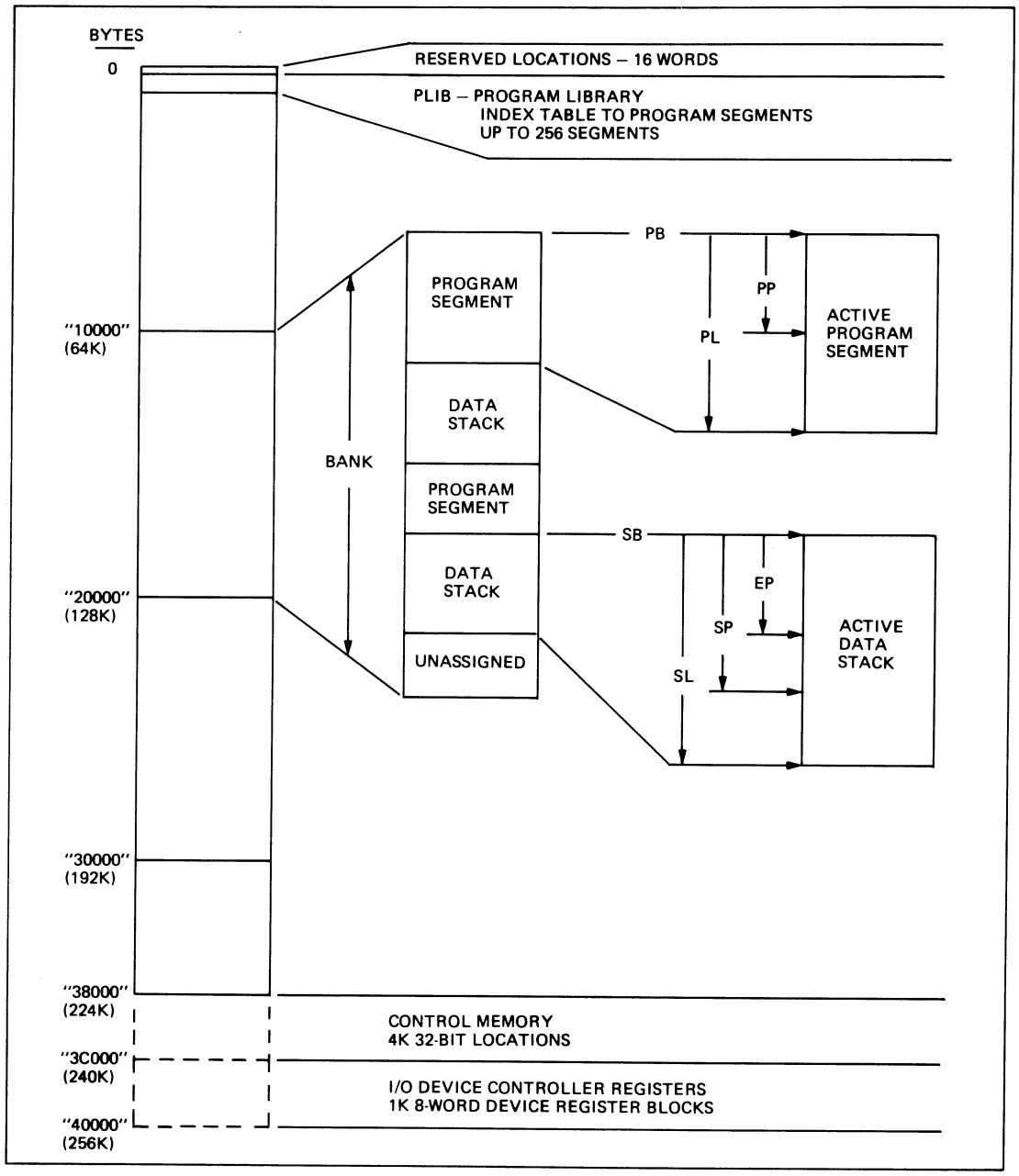


Figure A. Monobus Organization.

2.2 PROGRAM SEGMENT

The program segment contains the code generated for one or more MPL procedures, literal data and constant data, and an indirect address table to entry points within this procedure code. The program segment is specified by the three registers: Program Base (PB), Program Length (PL), and Program Pointer (PP).

The program segment contains the code corresponding to one or more MPL procedures. The number of external procedures contained in a segment is determined at program load time. The program segment is divided into a procedure section and a Program Reference Table, PRT. The procedure section contains the 32/S instructions, literal scalar data, and constant arrays. The PRT is an indirect address table to entry points within the procedure section. See Figure A.

The active program segment is defined by three registers:

PB Program Base 18 bits

This register points to the first word of the program segment; its least significant two bits are always zero.

PL Program Length 16 bits

This register points to the last word of the program segment relative to PB; its least significant two bits are always 10_2.

PP Program Pointer 16 bits

This register points to the next byte of 32/S instruction to be executed relative to PB.

The Program Reference Table, PRT, contains up to 256 16-bit addresses. Each address specifies the Program Base (PB) - relative location of the entry point for a procedure. Entries are made only for those procedures in the program segment which may be called by programs in another program segment. When a procedure is called (from a remote program segment), the PRT entry becomes the initial value of the Program Pointer (PP).

Entry addresses within the PRT are specified by a word-level index, PRTN:

Address of PRT entry address location = PB + PL - 2 * PRTN

The program segment may be of any length up to 64K bytes and may be positioned anywhere within main memory except that:

1. The segment may not cross the boundaries of a 65K byte memory bank;
2. The segment length is always a multiple of four bytes;
3. The segment always begins at an even doubleword address.

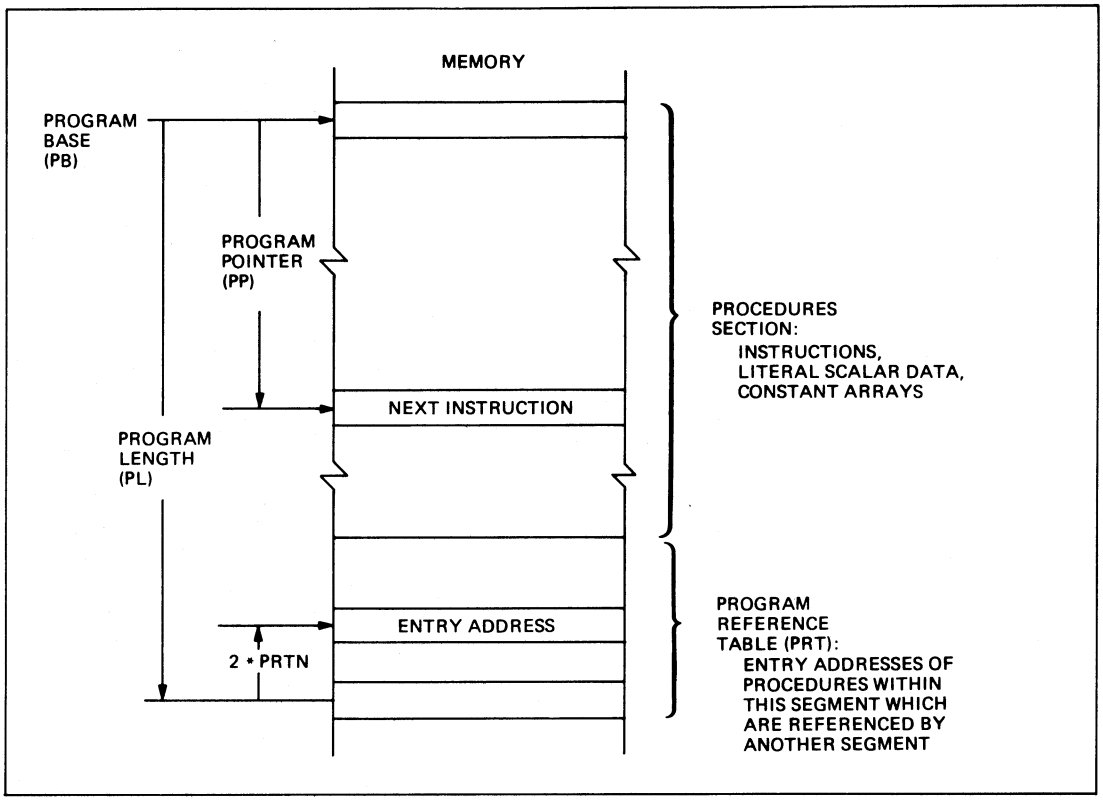


Figure A. Program Segment Format.

2.3 PROGRAM LIBRARY, PLIB

The PLIB provides program segment specifications which are automatically used when code in one program segment calls code in another program segment.

The Program Library, PLIB, is a table of pointers and other information which specifies all defined program segments. The PLIB starts at location "20" and provides a 2-word descriptor for up to 256 program segments. The entries in PLIB are generated by the loader and are used by the processor to invoke a new active program segment when a CALL, EXIT, or RESM instruction or an interrupt specifies to do so (see next topic).

The descriptor format is shown in Figure A. The first word of the descriptor is the value of Program Base, PB, to be used with that program segment. Specifically, it contains the most significant 16-bits of the 18-bit PB value. The lower two bits of the PB value are assumed to be zeroes; therefore, a program segment can only start on even doubleword boundaries. The second word of the descriptor contains a specification of the Program Length, PL, to be used with that program segment, a trace bit, T, and an attention bit, A.

The most significant 14-bits of the PL value are contained in the most significant 14-bits of the second word of the descriptor. The least significant 2-bits of the PL value are assumed to be 10_2; therefore, the program segment must be an integral number of doublewords in length.

The trace bit (bit 1 of the second word of the descriptor), indicates whether a trace interrupt is to be generated after execution of each 32/S instruction when this program segment is active. Specifically:

T = 0: no trace interrupt

T = 1: trace interrupt to be generated

The attention bit (bit 0 of the second word of the descriptor), indicates whether a PLIB attention interrupt is to be generated when the PLIB entry is accessed in the process of activating a new program segment (see next topic). Specifically:

A = 0: no attention interrupt

A = 1: attention interrupt to be generated

The attention bit set interrupt may be used by systems software as an indication that the program segment is absent from main memory, and that it must be read in from a back-up disc memory.

The two-word PLIB entries are specified by a double-word index, PLIBN, forward from byte location "20". An index of PLIBN = 0 specifies the first entry, which begins in memory address "20". Specifically:

Address of first word of PLIB entry = "20" + PLIBN * 4.

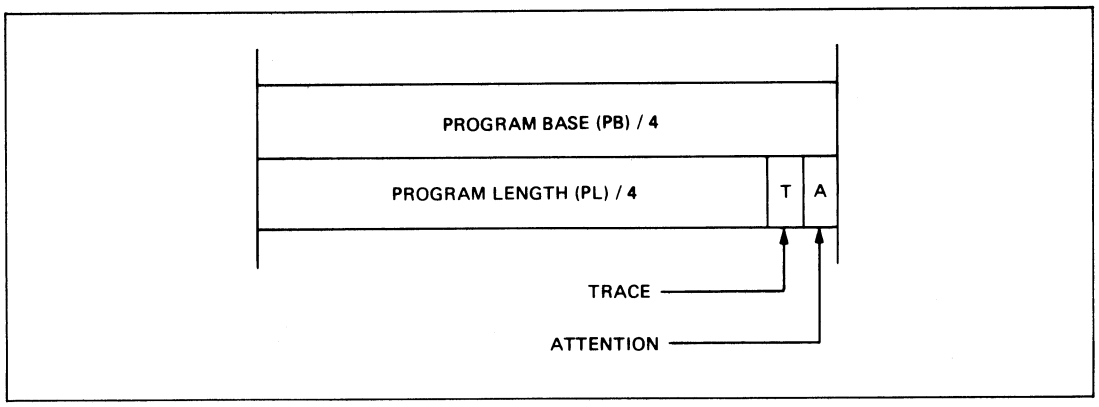


Figure A. Descriptor in Program Library (PLIB).

2.4 TRANSFER BETWEEN PROGRAM SEGMENTS

A new program segment is activated when the CALL or EXIT instruction being executed in the currently active program segment specifies a branch to code in another program segment. PLIB is used to locate and obtain the appropriate parameters for the new program segment.

When the procedure being executed in one program segment invokes a procedure which is in another program segment the control is switched to the new segment. PLIB is used in this process. The process is explained here with the aid of Figure A. Since the explanation involves a concept of the data stack and Mark entries in that data stack, the reader should expect to reread this topic after he has become familiar with these concepts from reading the following topics.

The 32/S code generated for a CALL statement in MPL consists of:

1. A MARK instruction to mark the start of the environment for execution of a new procedure in the data stack.

2. A sequence of instructions to push arguments for the called procedure into the stack on top of this Mark.

3. A CALL instruction to complete the Mark and to transfer control to the code for the new procedure.

If the called procedure is in a new program segment the MARK instruction will contain a PLIBN and PRTN entry. The PLIBN provides the entry point for the descriptor for the new program segment in PLIB. The PRTN is an index into the PRT of the new segment to obtain the entry pointer for the called procedure. The concluding CALL instruction then will obtain new PB and PL values from the descriptor in PLIB and will obtain a new PP value from the PRT of the new program segment. A Z bit value of 1 is used in both the MARK instruction and in a Mark entry to specify a change in program segments.

The A bit in the descriptor may specify that the called procedure cannot be executed. In this case, the called procedure is invoked and then immediately interrupted.

In more detail, the sequence is as follows:

1. The MARK instruction contains a bit (Z) which indicates if the called procedure is in a different program segment. If Z = 1, this instruction provides PLIBN and PRTN values for the called procedure. The MARK instruction begins the construction of a new Mark entry in the data stack and deposits Z, PLIBN, and PRTN in this mark. (If Z = 0, the called procedure is within the currently active segment, and the MARK instruction contains its entry address.)

2. Subsequent instructions push arguments for the called procedure on to the top of the stack, on top of the Mark whose construction was initiated by the MARK instruction.

3. The CALL instruction checks the value Z left for it by the MARK instruction in the Mark under construction in the data stack. If Z = 1, then the CALL instruction extracts the PLIBN value and uses it as an index into PLIB to obtain the descriptor for the new program segment. (If Z = 0, the CALL instruction extracts the entry address and loads it into PP.)

4. The CALL instruction extracts the PB and PL values from the descriptor and stores them into the PB and PL registers.

5. The CALL instruction then stores the current PLIBN (which it obtains from the Program Status Register) and the current PP value into the Mark being created in the stack.

6. At this point, the Mark being created in the stack has been completed and the CALL instruction adjusts the Environmental Pointer, EP, to point to this Mark.

7. The CALL instruction then tests the A bit in the program descriptor. An A value of 1 indicates that the called procedure cannot be invoked; for example, it could mean that the called program segment is absent from memory.

   If A is a 1, an attention interrupt is generated. An Interrupt Mark is pushed into the stack. This Mark contains the PRTN and PLIBN values for the called procedure; the Z bit is a 1 to indicate that the Mark's PP/PRTN entry is a PRTN value. The PLIBN of the called procedure is also pushed into the stack as an argument for the interrupt procedure. The process is then complete.

8. If A is a 0, the CALL instruction then tests the T bit in the program descriptor. If T is a 1, the internal trace status bit is set.

9. The CALL instruction then uses the PRTN value for the called procedure to index into the PRT of the new segment. It extracts the entry pointer from this PRT and loads it into the PP.

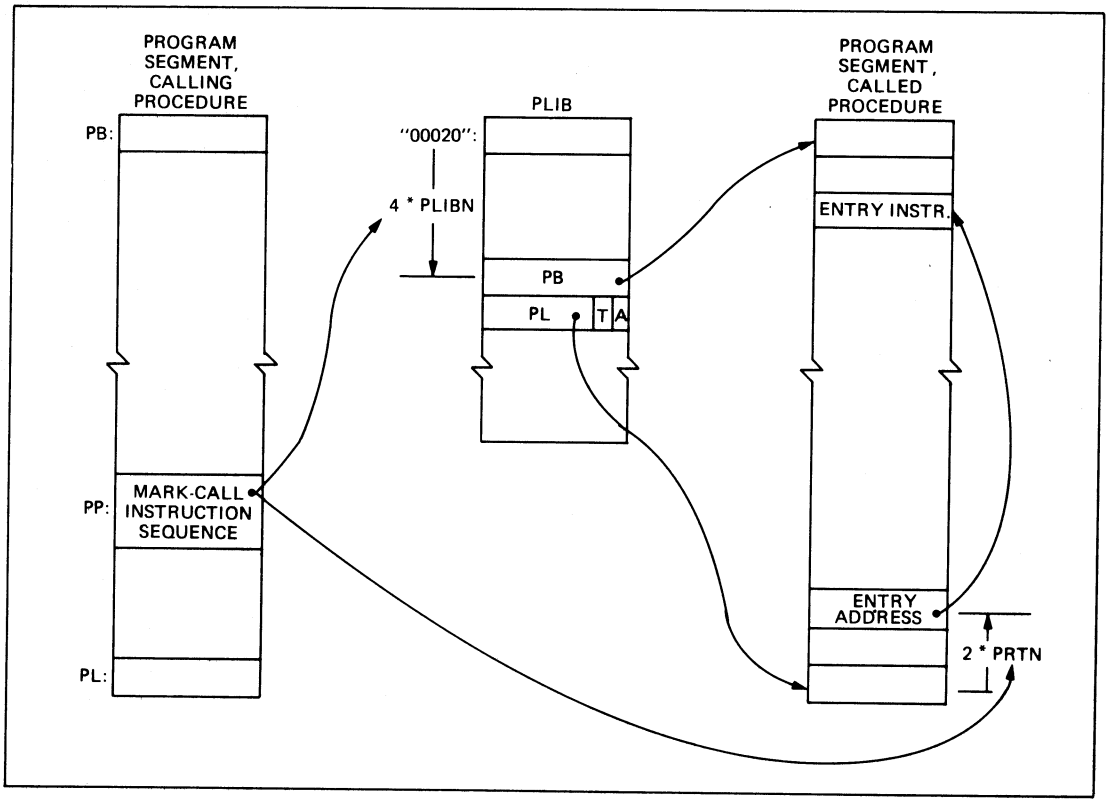


Figure A. Calling a Procedure in a Remote Program Segment.

2.5 DATA STACK

The data stack is the area of memory allocated for an individual machine user's data. It is a push-down stack which is specified by four registers; Stack Base (SB), Stack Length (SL), Environmental Pointer (EP), and Stack Pointer (SP). To gain a significant speed advantage, five hardware registers are provided as a dynamic head to the stack.

The data stack is the area of main memory allocated for the variable data belonging to an individual machine user. It is manipulated as a push-down stack, with five hardware "stack head" registers provided for use as the head, or top locations in the stack. There may be any number of data stacks allocated within main memory, but only one of these stacks is active at any given time. This topic defines the active stack format. (See topic 2.14 for the format of an inactive stack.) See Figure A.

Within the data stack, four-word entries call "Marks" are installed that delineate the location in the data stack where a new PROCEDURE block or BEGIN block began to use the data stack. (See topic 2.7.)

The top five word locations in the data stack are called TOS (top of stack) registers. Specifically:

TOS:	top of stack
TOS1:	location one level below top of stack
TOS2:	location two levels below top of stack
TOS3:	location three levels below top of stack
TOS4:	location four levels below top of stack

These names are used whether the particular TOS location exists in one of the stack head registers or exists in main memory. The TOS locations shift between main memory and the stack head registers dynamically as a program executes. The figure shows a typical situation in which the three topmost locations are in the hardware registers.

The active data stack is defined by four registers:

SB Stack Base — 18 bits

This register points to the first word of the data stack.

SL Stack Length — 16 bits

This register points to the last word in the data stack, relative to SB.

SP Stack Pointer — 16 bits

This register points at the top location of the data stack within main memory, relative to SB.

EP Environmental Pointer — 16 bits

This register points to the first word of the latest Mark within the data stack, relative to SB.

Since the data stack is word-oriented, the least significant bit of these registers is always a 0.

A data stack may be any length up to 64K and may be stored anywhere within main memory except that:

The data stack must be wholly located within a 64K memory bank.

NOTE: The process of reading the contents of SP, either via the front panel or via software debugging programs, always causes the contents of any active stack head registers to be placed into main memory and adjusts SP accordingly.

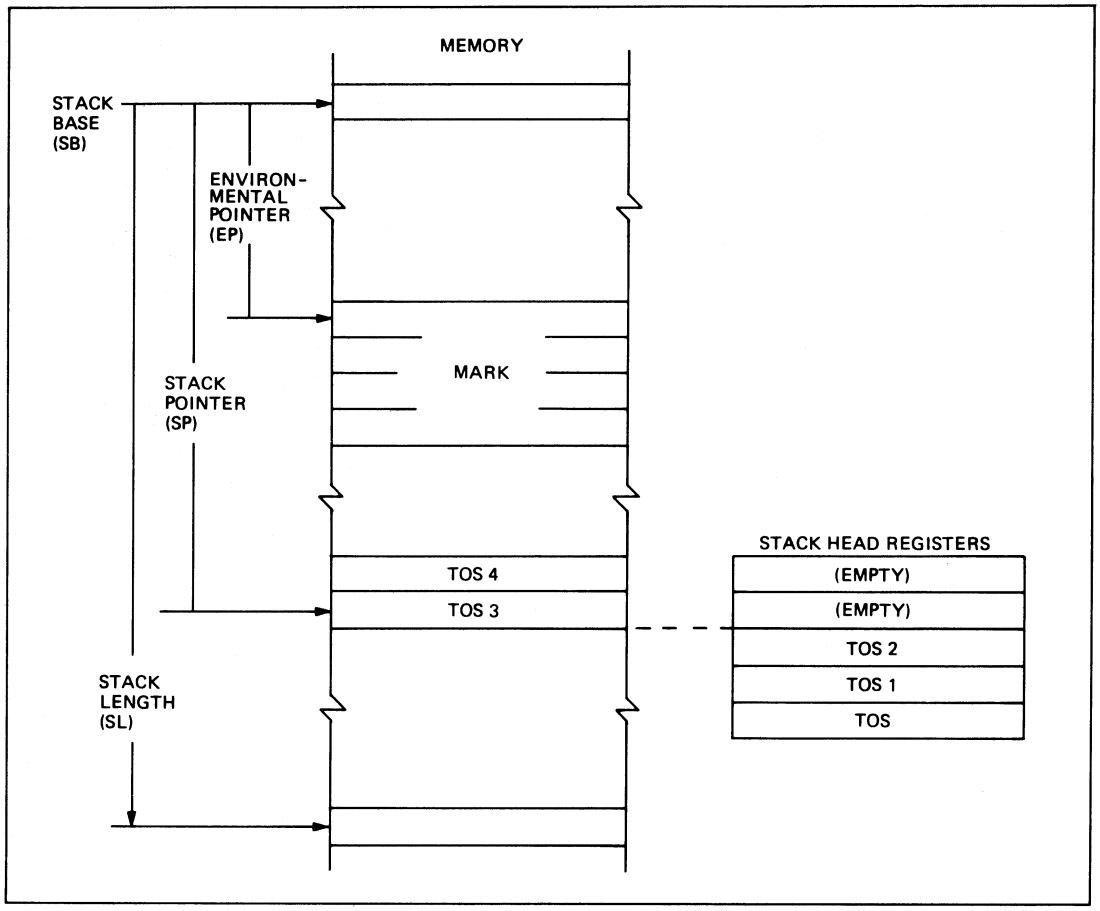


Figure A. Data Stack Format.

2.6 STACK HEAD REGISTERS

The stack head registers provide up to five levels of the top of the data stack and are automatically allocated to minimize the number of accesses to main memory in data stack operations.

The data stack head registers consist of five high-speed registers. The number of active stack head registers is variable. Hardware logic maintains a record of which stack head registers are empty and which one (if any) is the current top of the stack, so that data can be pushed into the stack or popped from the stack without transferring between registers. Most data stack operations as a result, actually can be performed within the 135 nanosecond clock time.

The processor operates in such a way as to use the stack head registers to minimize accesses to main memory. When data is pushed into the data stack it goes into stack head registers rather than into main memory - as long as there are any empty stack head registers. If the stack head registers are filled, or become filled during the push operation, the deepest entry in the registers overflows into main memory. Both situations are illustrated in Figure A. Note that the Stack Pointer, SP, always points to the highest stack location in main memory.

During the operation of popping data from the data stack, the processor pops data from the stack head registers without accessing main memory as long as sufficient data is available within the stack head registers. Data is not moved from main memory into the stack head registers during a pop operation. A pop operation occurring with the stack head empty simply causes the contents of the SP register to be decremented. No data is moved. See Figure B.

Note again that SP, the Stack Pointer, always points to the topmost location of the data stack within main memory.

NOTE: The process of viewing the contents of the Stack Pointer, SP, from the front panel automatically causes the contents of the stack head registers to be placed into main memory. This means that any attempt to see the data in the top of the data stack, by requesting a display of the location specified by the Stack Pointer, will automatically empty the stack head registers into memory, and therefore displays the actual top of the data stack. The programmer can normally ignore the five stack head registers and can visualize the stack as residing entirely in main memory.

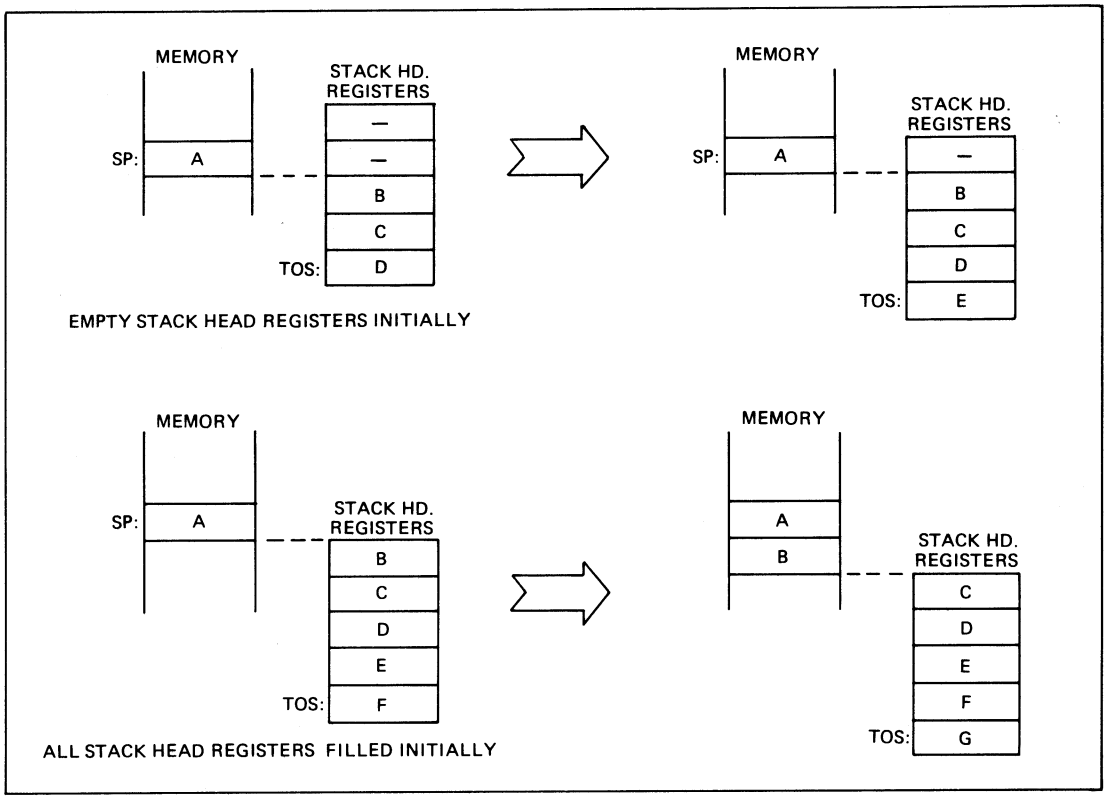


Figure A. Push Stack Operation.

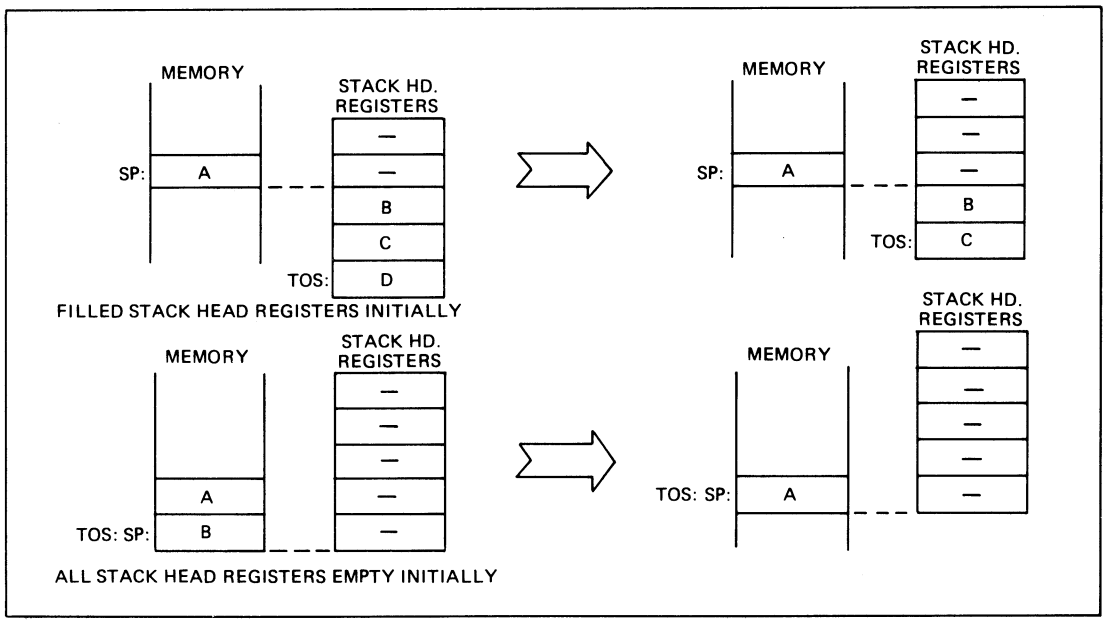


Figure B. Pop Stack Operation.

2.7 DATA STACK MARK

Four-word entries in the data stack, called Marks, delineate the stack environments which are local to each PROCEDURE or BEGIN block. They contain the information required to restore the data stack and program segment environment of the block previously being executed, and the information needed to access data local to an outer block.

As the code corresponding to an MPL PROCEDURE or BEGIN block is entered, the data stack is marked by a four-word entry called a Mark. This establishes the local reference point in the stack for the currently executing block and minimizes, thereby, the number of bits required for addressing data variables. As a block is exited the Mark is removed and the data stack is "rolled back" to the point preceding the Mark for that block. The first word of the latest Mark, installed when the current block was entered, is pointed to by the Environmental Pointer, EP. The area following a Mark, up to any subsequent Mark, is defined as the environment of that block. Figure A illustrates the data stack at a point within execution of an MPL program.

The first Mark (closest to the Stack Base) is for the MAIN PROCEDURE block of the program. The last Mark is for the block currently being executed. It is followed by the procedure arguments which were passed to the current block when it was entered, and the local variables which are being used by the currently executing block. The portion of the stack between the last Mark and the location specified by the Stack Pointer is the current environment.

Between the first and the current Marks are the Marks created for those blocks which have been entered but which have not been exited in the course of reaching the block currently being executed.

The general form of the Mark is illustrated in Figure A. The first two words of the Mark contain links, or pointers, to the Marks which delineate the start of preceding environments. These are used to access data local to a containing block and are used to unroll the stack as the current block is exited. The second two words contain the Program Pointer, Program Library Number and program status at the time that the block was entered. This information is required to return control to the calling, containing, or interrupted block, when this block is exited.

The Static Link, SLINK, points to the Mark of the next outermost block. Specifically, SLINK is the SB-relative address of the first word of the Mark for the next outermost MPL block. Note that the next outermost block is not necessarily the one in which the program was executing when the current block was entered. The "linked list" of SLINK entries within the Marks is used to access data which was declared in a block containing the current block. (See topics 2.10 and 2.11.)

The Dynamic Link, DLINK, points to the Mark of the block which was being executed when the current block was entered. Specifically, DLINK is the SB-relative address of the first word of the immediately preceding Mark. DLINK is used to roll back the data stack to the environment of the calling or containing block when the current block is exited.

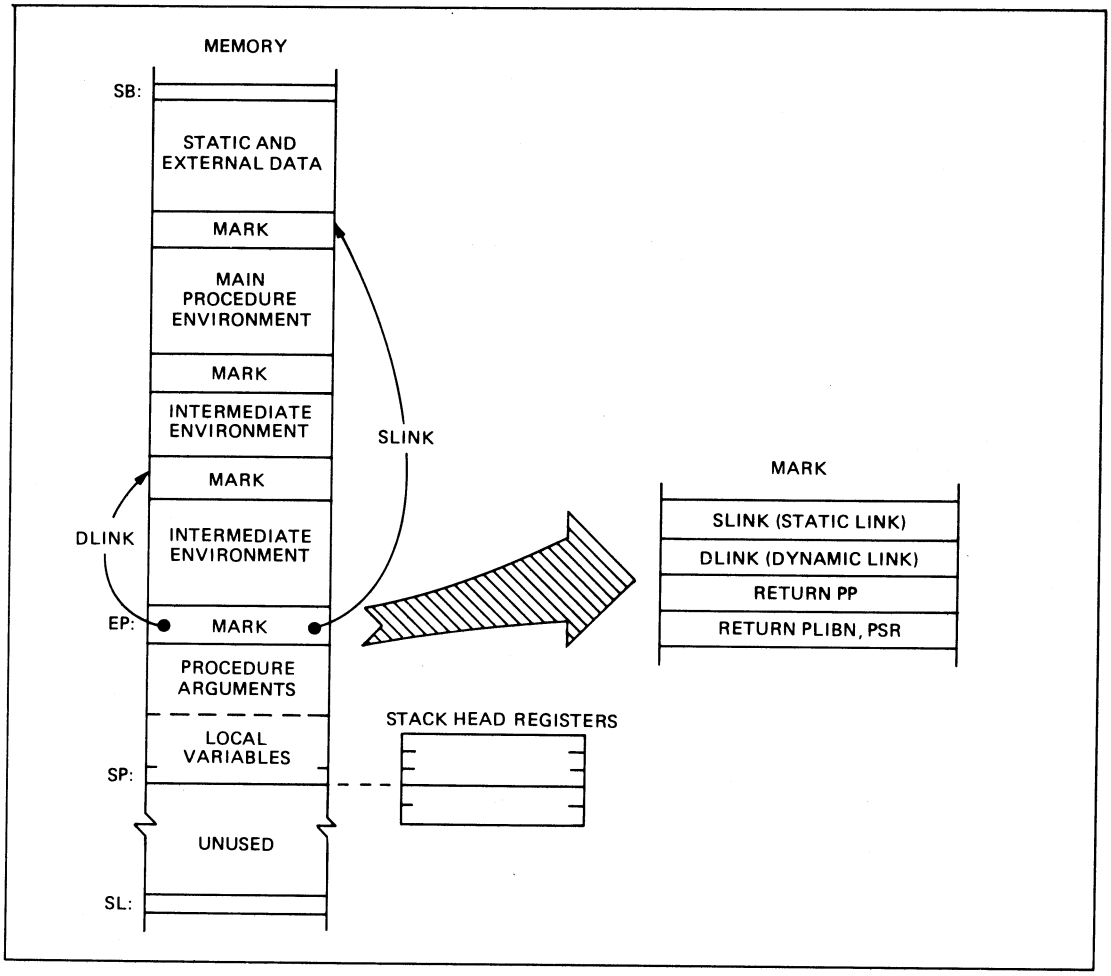


Figure A. Mark: Role in Data Stack and General Format.

2.8 MARK FORMATS

Formats are defined for the three types of Marks, Procedure, Begin, and Interrupt.

There are three types of Marks:

Procedure Mark:	created when a PROCEDURE block is entered;
Begin Block Mark:	created when a BEGIN block is entered;
Interrupt Mark:	created when the program being executed is interrupted.

Procedure Mark

The Procedure Mark is created whenever a new PROCEDURE block is entered, either as a result of executing an MPL CALL statement or as a result of encountering an MPL function. It is removed when the execution of the PROCEDURE block is completed and control is returned to the calling PROCEDURE or BEGIN block.

The first two words are the standard SLINK and DLINK entries. The last two words are the Program Pointer and Program Status Register values to be used when control is returned to the calling PROCEDURE or BEGIN block. Specifically:

SLINK Static Link — word 0

This word is the SB-relative address of first word of the Mark for the next outermost PROCEDURE or BEGIN block.

DLINK Dynamic Link — word 1

This word is the SB-relative address of first word of the Mark for the calling PROCEDURE or BEGIN block.

PP Program Pointer — word 2

This word is the Program Pointer value to be used for the next instruction to be executed upon return of control to the calling PROCEDURE or BEGIN block.

PLIBN Program Library Number — word 3, bits 15-8

This field is the index into PLIB for the calling PROCEDURE or BEGIN block. Upon exiting the current PROCEDURE, this PLIBN is used to retrieve the calling program segment (if it is remote) and is installed as the PLIBN field of the Program Status Register.

O Overflow — word 3, bit 2

This bit is the overflow bit of the Program Status Register at the time when the procedure (for which the Mark was created) was entered. Upon exiting the current procedure, the current Program Status Register overflow bit is ORed with the Mark's O bit and the resultant becomes the new overflow bit value in the Program Status Register (PSR). The effect is that if the PSR overflow was set either prior to entering the procedure, or during execution of the procedure, it will be set upon return to the calling block.

R Return word 3, bits 1, 0

This field is the number of words to be returned by the called procedure. At the point when the Mark is to be removed, the R words in the top of the stack are saved; the Mark is then removed and the R words are reinstalled in the top of the stack.

NOTE: A special form of the Procedure Block is defined for the MAIN PROCEDURE of an MPL program. It has a SLINK value of "0000" and a DLINK value of "FFFF".

Begin Mark

The Begin Mark is created whenever a BEGIN block is entered. It is removed when execution of the BEGIN block is completed.

The first word is the standard SLINK entry. The second word is a variation on the standard DLINK entry. The last two words are not filled in. (The return values of the Program Pointer, PLIBN, and Program Status Register which normally go into these two words are not needed because a BEGIN block is in-line code). Specifically:

SLINK Static Link word 0

This word is the SB-relative address of the first word of the Mark for the next outermost PROCEDURE or BEGIN block.

DLINK Dynamic Link word 1

If the DLINK of the immediately preceding Mark is odd-valued, this word is a copy of that DLINK; otherwise this word is the SLINK value (for this Mark) plus 1.

The DLINK entry is designed to facilitate the processor's task of removing a sequence of one or more Begin Marks if execution of an outer PROCEDURE block terminates before execution of the BEGIN blocks is completed. DLINK and SLINK entries for Procedure Marks, and SLINK entries for Begin Marks are word addresses and are therefore even-valued. The DLINK entries in Begin Marks, however, are odd-valued and therefore identify these Marks. Furthermore, the DLINK entry of a Begin Mark specifies the address of the next outer Procedure Mark.

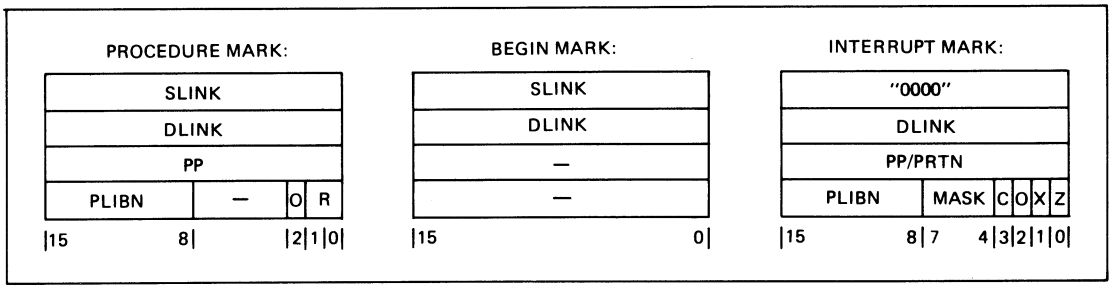


Figure A. Mark Formats.

2.9 MARK FORMATS (Continued)

Formats are defined for the three types of Marks, Procedure, Begin, and Interrupt.

Interrupt Mark

The Interrupt Mark is created when an interrupt has been honored; the Interrupt Vector Table has been accessed (see topics 3.1 and 3.2), and processing of the interrupt procedure is to begin. It is removed after execution of the interrupt procedure is completed.

The standard SLINK entry, normally the first word, has no meaning since there is no next outer block (in the interrupted program segment) for the interrupt PROCEDURE block. The second word, however, is the standard DLINK entry which points to the Mark of the interrupted block. The last two words normally are the Program Pointer and Program Status Register values to be used when control is returned to the interrupted block. An exception to this is when the interrupt is generated as a result of a call to a procedure in a remote program segment which cannot be executed because the PLIB entry for that remote program segment has its attention bit (A) set. In this latter case, the third word contains the PRTN value for the desired entry point into the called procedure. (This transfer between program segments is discussed in topic 2.4.) Specifically:

SLINK — Static Link — word 0

This word is set to "0000", identifying this as an Interrupt Mark.

DLINK — Dynamic Link — word 1

This word is the SB-relative address for the first word of the Mark for the interrupted block.

PP/PRTN — Program Pointer/Program Reference Table Number — word 2

If Z = 0, this word is the Program Pointer value to be used for the next instruction to be executed upon return of control to the interrupted block.
If Z = 1, this word is the PRTN value to enter the called procedure after the attention interrupt has been processed.

PLIBN — Program Library Number — word 3, bits 15-8

This field is the index into PLIB for the interrupted block. Upon returning control to the interrupted procedure, this PLIBN is used to retrieve the interrupted program segment, and is installed as the PLIBN field of the Program Status Register.

MASK — Interrupt Mask — word 3, bits 7-4

This field is the interrupt mask for the interrupted block. Upon returning control to the interrupted procedure, this interrupt mask is installed as the interrupt mask field of the Program Status Register.

C — Carry — word 3, bit 3

This bit is the carry bit of the Program Status Register at the time the block was interrupted. Upon returning control to the interrupted procedure this C bit is installed as the carry bit in the Program Status Register.

O — Overflow — word 3, bit 2

This bit is the overflow bit of the Program Status Register at the time the block was interrupted. Upon returning control to the interrupted procedure this bit is installed as the overflow bit in the Program Status Register.

X — Mode — word 3, bit 1

This bit is the normal/executive mode bit of the Program Status Register at the time the block was interrupted. Upon returning control to the interrupted procedure, this X bit is ANDed with the X bit in the current PSR. This ANDing operation prevents an interrupt procedure running in the normal mode from returning control to a previous environment in executive mode.

Z — word 3, bit 0

This bit specifies the meaning of the third word. If Z = 1, the interrupt was caused by a set attention bit in a PLIB entry when a procedure in a remote program segment was called; the third word is then the PRTN for that called procedure. If Z = 0, the interrupt is due to any one of the other types of interrupts and the third word is then the Program Pointer value to use when control is returned to the interrupted procedure.

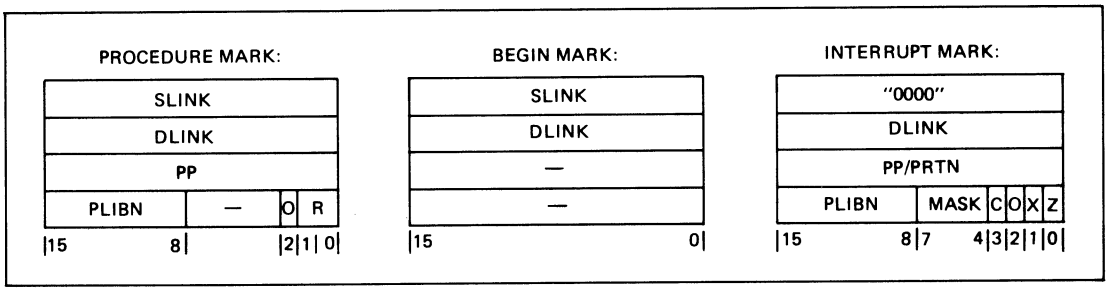


Figure A. Mark Formats.

2.10 DATA STACK ENVIRONMENTS & MPL PROGRAM STRUCTURE

This topic states the relationship between the structure of an MPL program and the environments, delineated by Marks, within a data stack.

MPL programs are statically structured of PROCEDURE and BEGIN blocks, with block within block, and with block following block. MPL programs are dynamically executed in a sequence determined not only by the static structure, but also by the CALL statements which jump the control from inside one block to the beginning of a block external to the one containing the statement. As program execution progresses, the data stack is marked off into environments by the Mark entries. The SLINK and DLINK entries in these Marks link these environments on the basis of their static and their dynamic relationships.

Figure A illustrates an example of an MPL program and the data stack at a point in time during execution of that program. The MPL program consists of a MAIN PROCEDURE A which contains a BEGIN block B and a PROCEDURE block D. Blocks B and D each contain BEGIN blocks. PROCEDURE D is called from inside BEGIN block C. The dynamic sequence of execution of the blocks is shown below the MPL program.

The data stack is shown during the time that code for BEGIN block E is being executed. The SLINK and DLINK entries are shown in each Mark. The Environmental Pointer, EP, values, which were current at the time each Mark was the latest one in the stack, are shown at the base of each Mark.

The first Mark is for the MAIN PROCEDURE. It is a special case of the Procedure Mark, containing a "0000" SLINK and an "FFFF" DLINK.

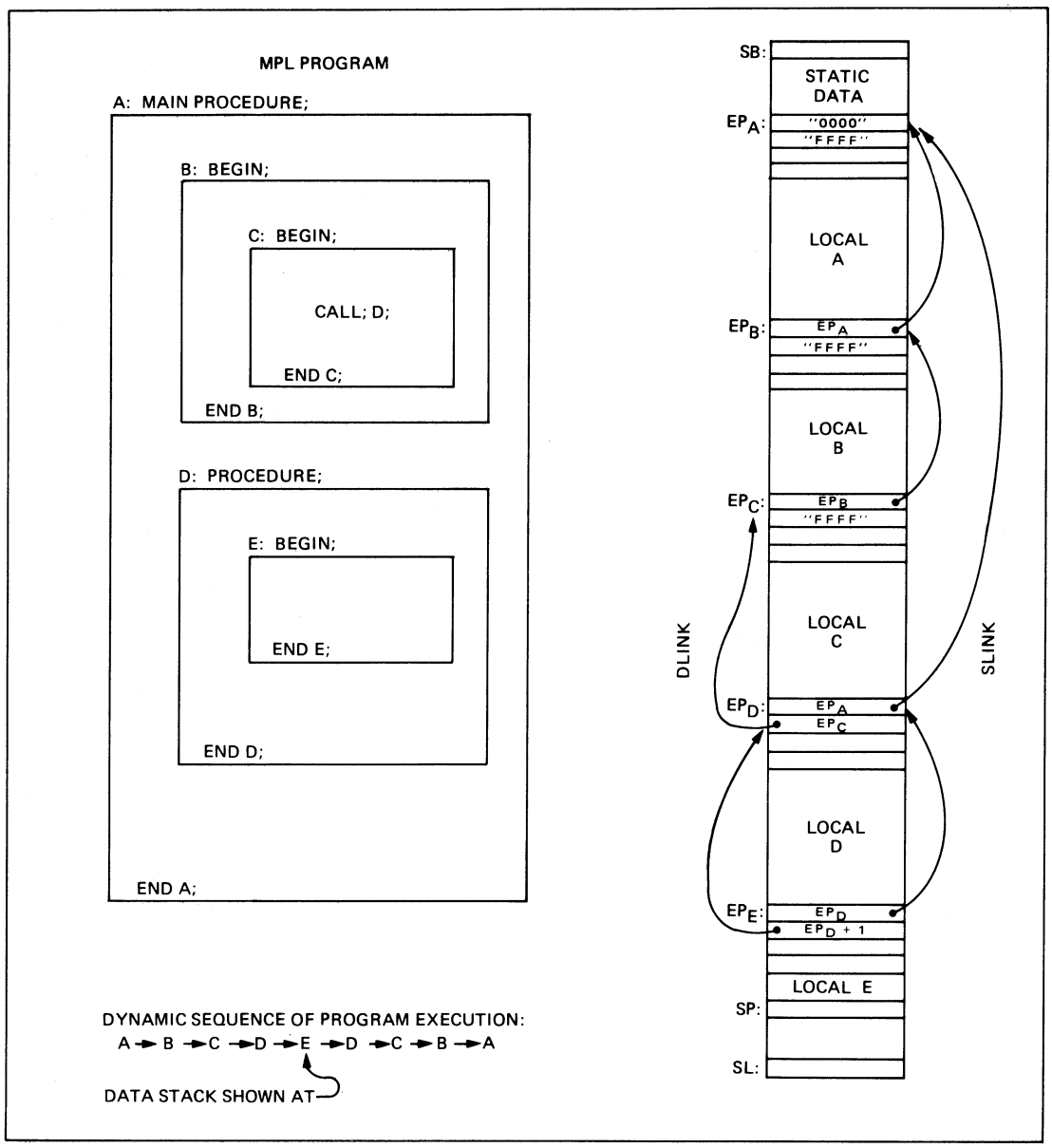


Figure A. Relationship Between MPL Program Structure and Data Stack Environments.

2.11 THE DELTA LEX-LEVEL CONCEPT

The Delta Lex-Level, DLEX, is a number which specifies the static relationship between blocks in the MPL program. The concept is introduced to simplify the explanation of various 32/S operations.

The Delta Lex-Level, DLEX, is an abbreviation for the difference in lexographical levels. Figure A shows an MPL program graphically. The DLEX parameter between CALL statements and the procedure which they call is shown to the right of each CALL.

From the examples shown in Figure A, it can be seen that:

DLEX = 0: a call to a procedure which is declared at the same level as the CALL statement; the contents of the called PROCEDURE block itself is therefore one level down, a "son."

DLEX = 1: a call to a procedure which is declared at one level outside the CALL statement; the contents of the called PROCEDURE block itself is therefore at the same level, a "brother."

DLEX = 2: a call to a procedure which is declared two levels above the CALL statement; the contents of the called PROCEDURE block itself is therefore one level above, a "father" (C PROCEDURE), or an "uncle" (B PROCEDURE).

Up to 16 levels of DLEX may be used. (The limitation is the size of the DLEX field in the MARK and LADR instructions.)

Figure A also illustrates two illegal CALL statements; a call to PROCEDURE D and a call to PROCEDURE F, both from PROCEDURE A. These CALL statements are illegal because the names 'D' and 'E' are not known in PROCEDURE A. See the MPL reference manual for an explanation of naming scope.

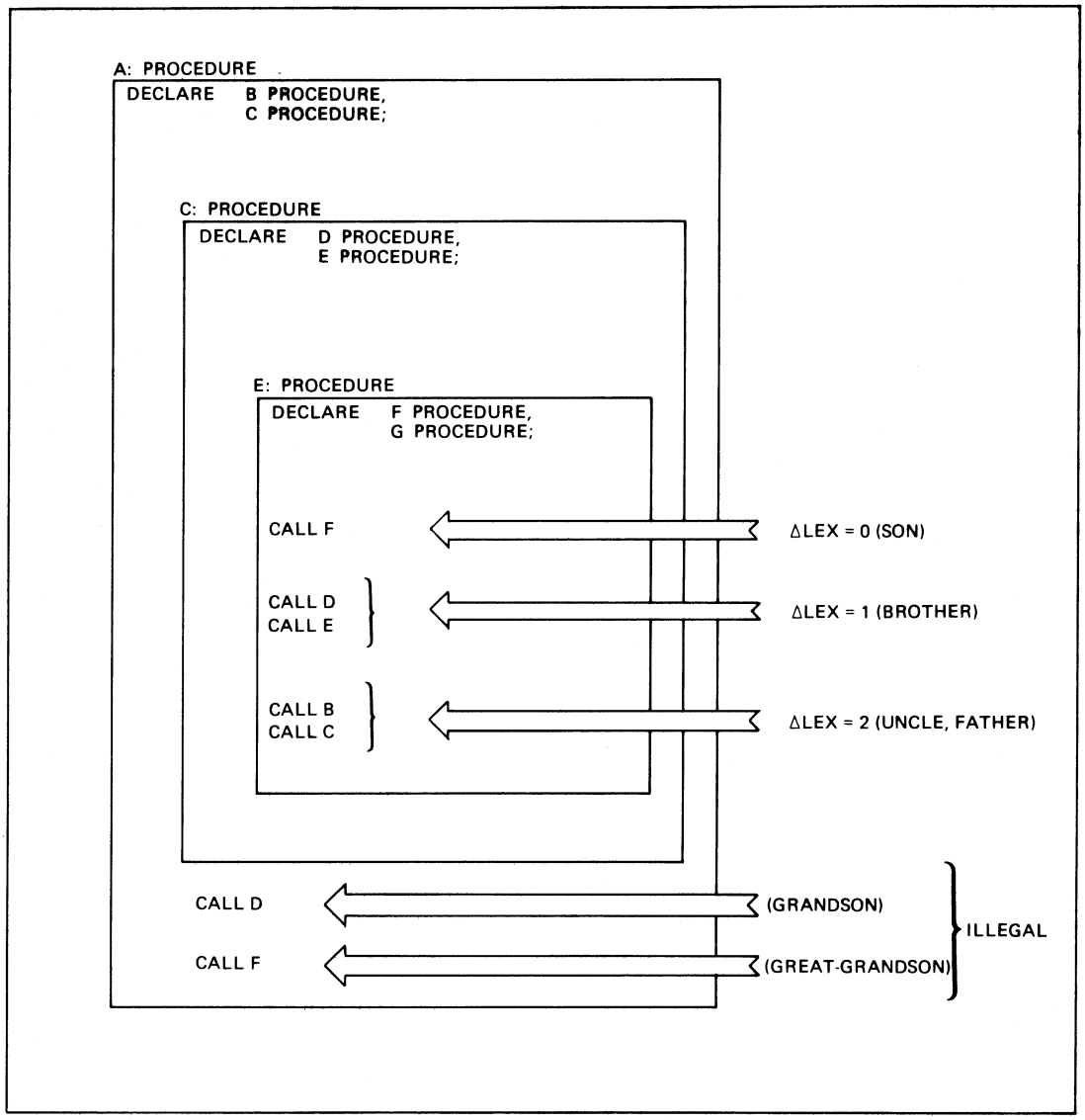


Figure A. Delta Lex Level (ΔLEX) Concept.

2.12 DATA STACK TRANSFORMATIONS - BEGIN BLOCK EXECUTION

The BEGIN and END statements compile into the BENT and EXIT instructions which create and remove, respectively, the BEGIN mark in the data stack.

Figure A shows the sequence of states of the data stack as a BEGIN block is initiated, executed, and exited.

The data stack immediately before execution of the BEGIN block is shown in the upper left. (Two top entries, TOS and TOS1, are shown in the stack head registers as a typical situation.)

The BEGIN statement compiles into a Begin Entry, BENT, instruction (see topic 9.1). The BENT pushes the contents of active stack head registers into the stack in main memory. It then creates the BEGIN Mark and adjusts the EP and SP pointers.

Execution of the BEGIN block code then uses the data stack environment above the latest, Begin, Mark. A typical data stack at the completion of execution of the BEGIN block code is shown in the lower right of the figure.

The END statement for a BEGIN block compiles into a Begin Exit, BXIT, instruction (see topic 9.1). The BXIT "rolls back" the data stack to the environment immediately preceding the latest, Begin, Mark. It does this by adjusting the EP to the base of the previous Mark (using SLINK) and adjusting the SP to the location preceding the base of the removed Begin Mark.

Note that, since a BEGIN block returns no values to the preceding environment, all data in the stack above the removed Begin Mark is lost.

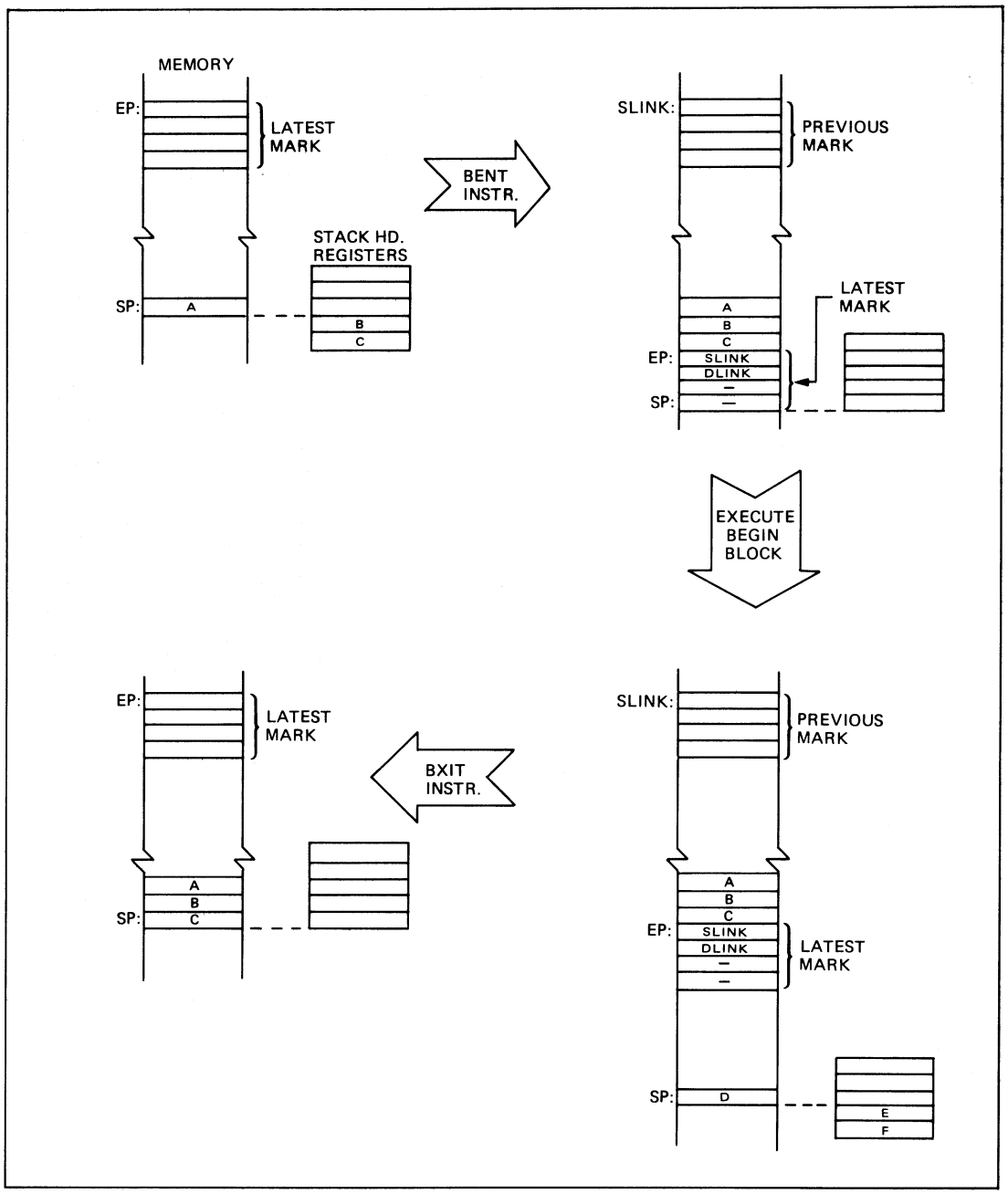


Figure A. Data Stack as Enter, Execute and Exit Begin Block.

2.13 DATA STACK TRANSFORMATIONS - PROCEDURE BLOCK EXECUTION

The PROCEDURE statement compiles into MARK and CALL instructions which create the Procedure Mark and the END and RETURN statements compile into an EXIT instruction which removes the Procedure Mark.

Figure A shows the sequence of states of the data stack as a PROCEDURE block is initiated, executed, and exited.

The data stack, immediately before execution of the PROCEDURE block, is shown in the upper left. (Two top entries, TOS and TOS1, are shown in the stack head registers as a typical situation.)

The PROCEDURE statement compiles into a MARK instruction, followed by code which pushes arguments into the stack, followed by the CALL instruction. The MARK instruction pushes the contents of the active stack head registers into the stack in main memory. It then initiates the creation of the Procedure Mark. The Stack Pointer, SP, advances accordingly during this operation, but, since the Procedure Mark is not yet completed, the Environmental Pointer, EP, still points to the same latest active Mark. (See topic 9.2)

32/S code following the MARK instruction pushes the arguments for the called procedure into the data stack. The data stack then looks typically as shown in the upper right of the figure.

The CALL instruction then completes the Procedure Mark and adjusts the Environmental Pointer, EP, to (and thus to activate) this Mark. (See topic 9.3.) The data stack then looks typically as shown in the lower right of the figure.

The called procedure then is executed, resulting in a data stack which looks typically as shown in the middle lower portion of the figure.

The END and RETURN statements compile into an EXIT instruction which "rolls back" the data stack to the environment immediately preceding the Procedure Mark. (See topic 9.4.) It does this by adjusting the Environmental Pointer, EP, (using DLINK) to point to (and thus activate) the previous Mark in the data stack.

The removed Procedure Mark contains the parameter R, the number of data words to be returned to the previous environment. The EXIT instruction saves the topmost R data words by leaving them in the stack head registers. It eliminates other data belonging to the exited procedure by adjusting the Stack Pointer, SP, to the location immediately preceding the removed Procedure Mark.

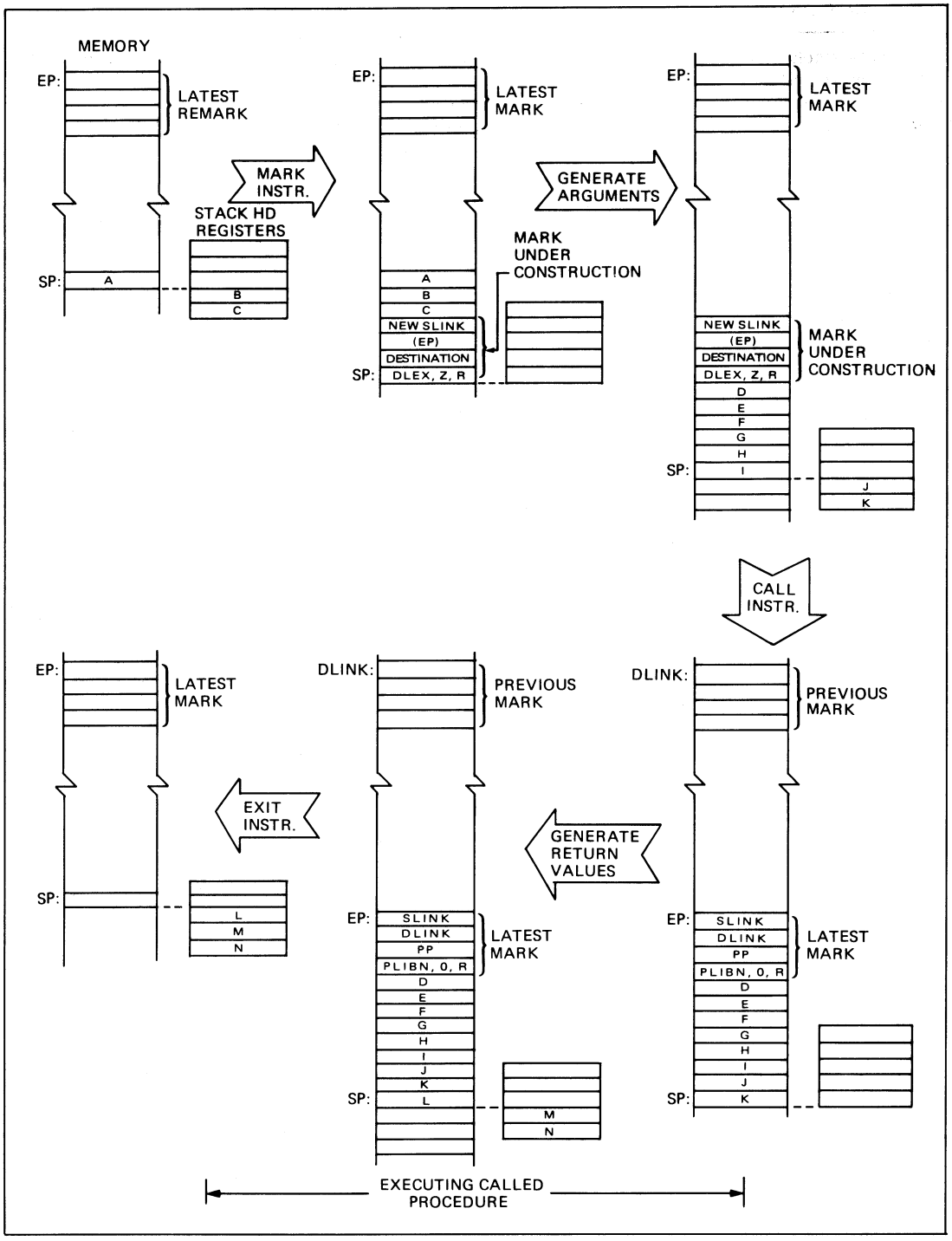


Figure A. Data Stack as Call, Execute, and Exit Procedure.

2.14 INACTIVE DATA STACK

Inactive data stacks are "capped off" by an Interrupt Mark and cataloged in a Stack Descriptor table which is maintained by system software.

At any given time, only one data stack in memory is active. All other data stacks are defined as inactive. The format of an inactive data stack is shown in Figure A.

The base location and length of each inactive stack is maintained, by system software, as a two-word descriptor. When an inactive data stack is activated the first word of this descriptor is loaded in the Stack Base, SB, register and the second word is loaded in the Stack Length, SL, register.

The Stack Pointer, SP, value for the data stack, at the time when it was made inactive, was stored in the first word of the stack. When an inactive data stack is activated this SP value is loaded into the Stack Pointer register.

The top of the inactive data stack contains an Interrupt Mark. This Mark is usually created by an interrupt which caused the data stack to become inactive (and simultaneously activated another stack).

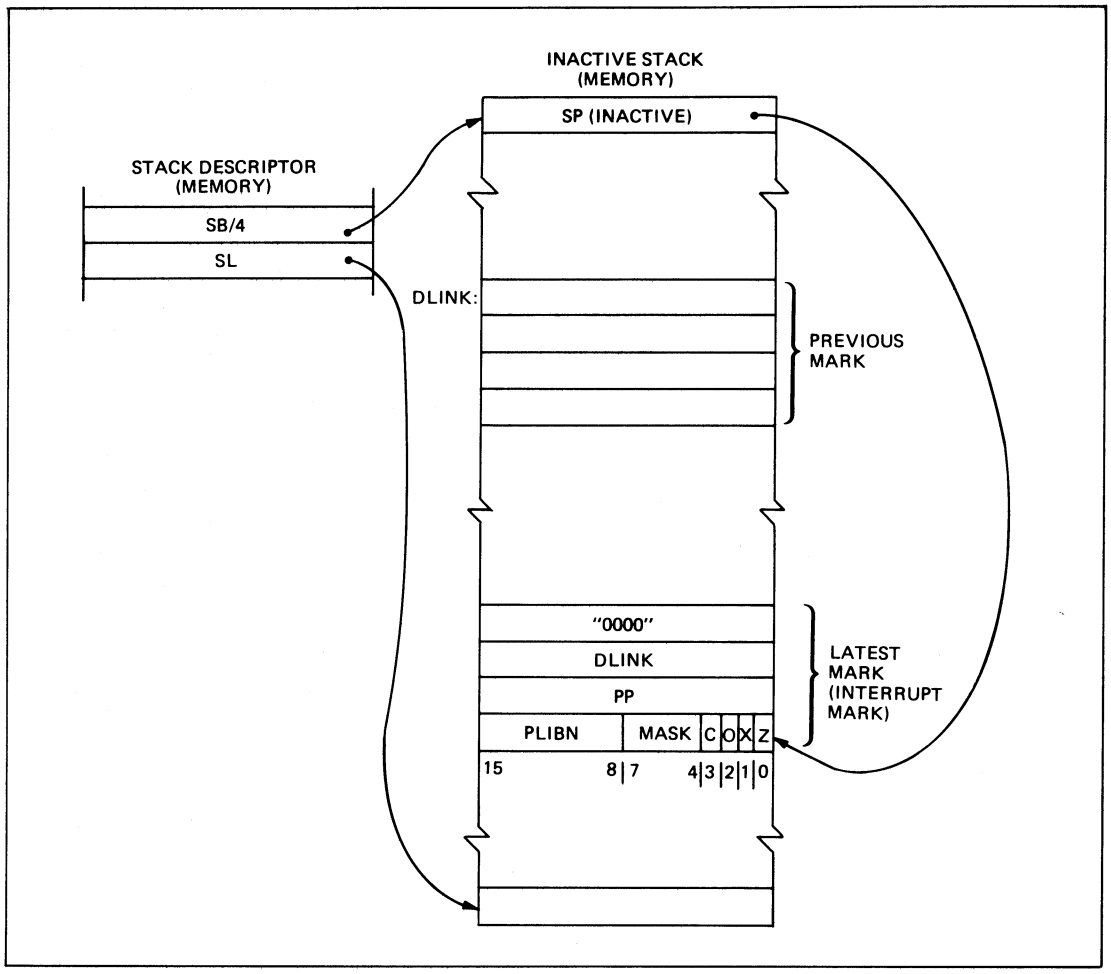


Figure A. Inactive Data Stack Format.

2.15 RESERVED MEMORY LOCATIONS

The first 32 locations of main memory are reserved for special purposes. The following locations are reserved for the variable-length Program Library, PLIB.

The processor utilizes the first 18 byte locations to store pointers and other information. The next 14 bytes are reserved for a second 32/S processor. The Program Library, PLIB, begins in the next location. The reserved memory locations are defined in Figure A.

Interrupt Stack Base Address — location: "00"

This word is the upper 16 bits of the 18 bit base address of the interrupt stack. The low order two bits of the address are zeroes. The contents of this location are transferred to the SB register when the interrupt stack is activated.

Interrupt Stack Length — location: "02"

This word is the length of interrupt stack (number of bytes).

Timer — locations: "04" and "06"

These two words are a 32 bit counter which is incremented by the processor real-time clock. A carry out of the most significant bit in location "04" causes a timer overflow interrupt (see topic 3.3).

Base Address of Concurrent I/O Control Block Table — location: "08"

This word is the upper 16 bits of the 18 bit base address of the CIOCB table. The low order bits of the address are always zeroes.

Base Address of Interrupt Vector Table — location: "0A"

This word is the upper 16 bits of the 18 bit base address of the Interrupt Vector Table. The low order two bits of the address are always zeroes.

Maximum Number of Devices — location: "0E"

This word is the value used to verify a device number when an external input/output interrupt occurs. A device number less than this value will be processed as a normal external interrupt. A device number greater than or equal to this value will cause a maximum device number exceeded interrupt. (See topic 3.3.) Note that this value is the maximum legal device number plus one.

Number of Entries in Concurrent I/O Control Block Table — Location: "10"

This word is the actual number of entries contained in the CIOCB table and is used to verify that the device requesting service has a CIOCB to control it. An illegal device number will cause the process to immediately halt to indicate a fatal error condition (see topic 3.3.)

LOCATION		DESCRIPTION
DECIMAL	HEXIDECIMAL	
0	00	INTERRUPT STACK BASE ADDRESS / 4
2	02	INTERRUPT STACK LENGTH
4	04	TIMER (UPPER 16 BITS)
6	06	TIMER (LOWER 16 BITS)
8	08	CONCURRENT I/O CONTROL BLOCK BASE ADDRESS / 4
10	0A	INTERRUPT VECTOR TABLE BASE ADDRESS / 4
12	0C	RESERVED
14	0E	MAXIMUM I/O DEVICE NUMBER, PLUS ONE
16	10	NUMBER OF ENTRIES IN CONCURRENT I/O CONTROL BLOCK
18	12	RESERVED
20	14	
22	16	
24	18	
26	1A	
28	1C	
30	1E	
32	20	FIRST ENTRY IN PROGRAM LIBRARY (PLIB)

Figure A. Reserved Memory Locations.

2.16 PROGRAM STATUS REGISTER

The Program Status Register, PSR, maintains the status of the currently executing program. It consists of 16-bits of status which may be interrogated from the Maintenance Panel.

The status of the currently executing program is recorded in 19 bits. The Program Status Register, PSR, as shown on the Maintenance Panel, contains 16 bits of information, organized as shown in Figure A. The three remaining status bits are external to the PSR (see topic 2.17).

The definitions of the Program Status Register fields are as follows:

PLIBN Program Library Number bits: 15-8

This field is the index into the Program Library, PLIB, for the currently executing program segment.

MASK Interrupt Mask bits: 7-4

This field is the enable/disable mask for the four external interrupt lines. This mask specifies which of these four lines are to have external interrupt requests acknowledged. This field may be modified under software control by executing an XIM instruction (see topic 9.11).

The definition of these bits is:

bit = 1: interrupt enabled

bit = 0: interrupt disabled

The bit positions are defined as:

bit 4: operator interrupt

bit 5: timer interrupt

bit 6: external interrupt line 0

bit 7: external interrupt line 1

C Carry bit: 3

This field is the carry status. It is set, or reset, as a result of execution of various 32/S instructions and tested by the TCAR instruction (see topic 9.11). Specifically:

C = 1: carry set

C = 0: carry reset

O Overflow bit: 2

This field is the overflow status. It is set as a result of execution of various 32/S instructions and tested and reset by the TOVF instruction (see topic 9.11). Specifically:

O = 1: overflow set

O = 0: overflow reset

X Mode bit: 1

This field specifies whether the program is running in executive or normal mode. It is set, or reset, by RESM, IXIT instructions, a restart, or an interrupt. Specifically:

X = 1: executive mode

X = 0: normal mode

The executive mode permits execution of privileged instructions (PNOP, XIM, MICR, and RESM) and use of the absolute memory addressing mode; neither is permitted in the normal mode. Certain protect checks are not made in Executive mode.

W Wait State bit: 0

This field specifies whether the program is in the wait mode or in the running mode. It is set by a WAIT instruction and reset by an interrupt. Specifically:

W = 1: wait state

W = 0: non-wait state

In the wait mode, instructions are not executed, but the timer and concurrent I/O operate.

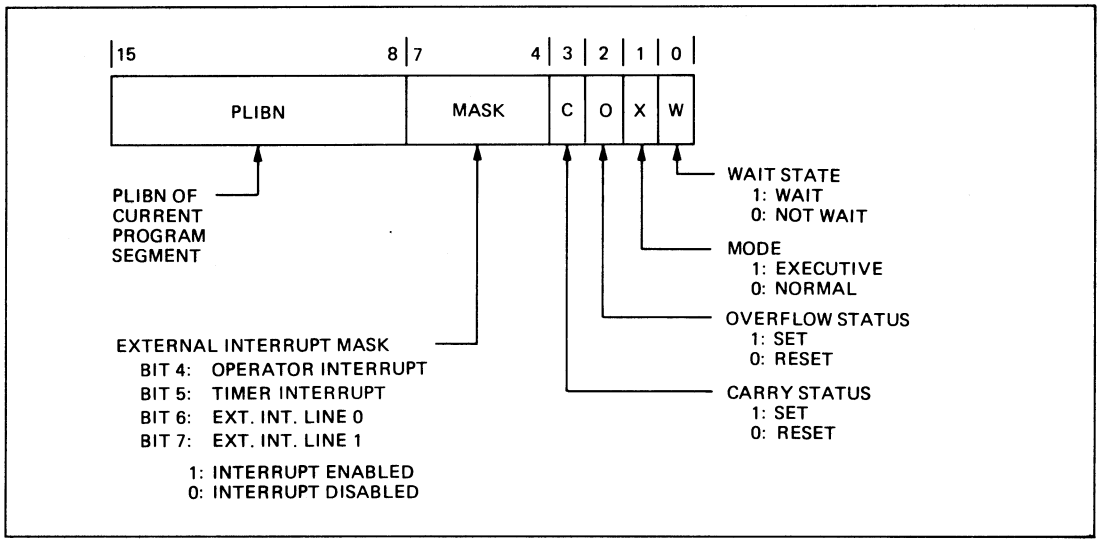


Figure A. Program Status Register (PSR) as Displayed on Maintenance Front Panel.

2.17 STATUS BITS EXTERNAL TO THE PSR

The processor utilizes three status bits which are external to the Program Status Register.

Three status bits are stored external to the Program Status Register, and are not accessible from the front panel. Specifically:

T — Trace — 1 bit

This bit specifies whether or not an internal trace interrupt is to be generated after execution of each instruction. It is set by activating a procedure segment whose PLIB trace bit is set. The trace status is reset when the trace interrupt occurs. Specifically:

T = 1: generate internal trace interrupt after execution of next 32/S instruction.

T = 0: do not generate trace interrupts.

PI — Postpone Interrupt — 1 bit

This bit specifies whether or not all external and internal interrupts are to be disabled until after execution of the next 32/S instruction. It is set when a traceable segment is activated and when an XIM instruction is executed and is reset by any 32/S instruction execution. Specifically:

PI = 1: disable all interrupts until after execution of next 32/S instruction.

PI = 0: do not disable all interrupts.

ISA — Interrupt Stack Active — 1 bit

This bit specifies whether or not the currently active data stack is the interrupt stack. It is set by an interrupt to the Interrupt Stack and by a restart. It is reset by a RESM instruction. Specifically:

ISA = 1: current stack is the interrupt stack.

ISA = 0: current stack is not the interrupt stack.

3 Interrupts

3.1 INTERRUPT ARCHITECTURE

The 32/S provides both internal and external interrupt facilities. The latter include four external interrupt lines for use by I/O device controllers. An interrupt mask enables/disables interrupts on two of these lines, the operator interrupt and the timer overflow interrupt. A unique interrupt vector is provided for each interrupt source, including each unique device number. Each interrupt vector provides an armed/disarmed state bit.

Interrupts are divided into two classes: internal and external. Internal interrupts are generated as a result of executing certain 32/S instructions (e.g., Supervisor Call). External interrupts are generated by events other than instruction execution.

External interrupts generated by I/O device controllers are assigned to one of four external interrupt lines (by logic within the controller). The operator interrupt button, timer overflow, power fail and maximum device number exceeded are unique external interrupt sources.

Internal interrupts are always enabled.

External interrupts generated by a standard I/O device controller can be enabled or disabled within the controller by setting mode bits within the controller. In addition, two of the external interrupt lines (line 0 and 1) can be individually enabled or disabled by two corresponding mask bits within the Program Status Register. The other two external interrupt lines are always enabled.

A unique interrupt vector number is associated with each source of internal interrupt and each source of external interrupt, including each I/O device controller number. The interrupt vector number indexes into an Interrupt Vector Table. The entries in this table specify the interrupt procedure, and the stack environment and Program Status Register to be used with that procedure.

The Interrupt Vector Table entry includes a bit which specifies whether the interrupt is armed or disarmed. The interrupt procedure is only invoked if the interrupt is armed. The interrupt is disregarded if it is disarmed.

The request for an internal interrupt arises as a result of a 32/S instruction execution. If the Interrupt Vector Table entry specifies the interrupt to be armed, the instruction execution sequence concludes with the set up of the environment for processing of the interrupt. The interrupt is then processed.

The requests for external interrupts arise asynchronously with 32/S instruction execution. External interrupts are serviced between instruction executions. Each external interrupt source has an assigned priority. If more than one enabled pending interrupt exists at the conclusion of an instruction execution the highest priority one is honored. External interrupt priorities are specified in topic 3.4.

If the Interrupt Vector Table entry for an honored interrupt is armed the processor sets up the environment for processing the interrupt. The interrupt is then processed. If it is disarmed and one or more other external interrupts are pending, the next highest priority armed interrupt is then honored. If it is disarmed and no other external interrupt requests are pending, processing continues with the next 32/S instruction in sequence.

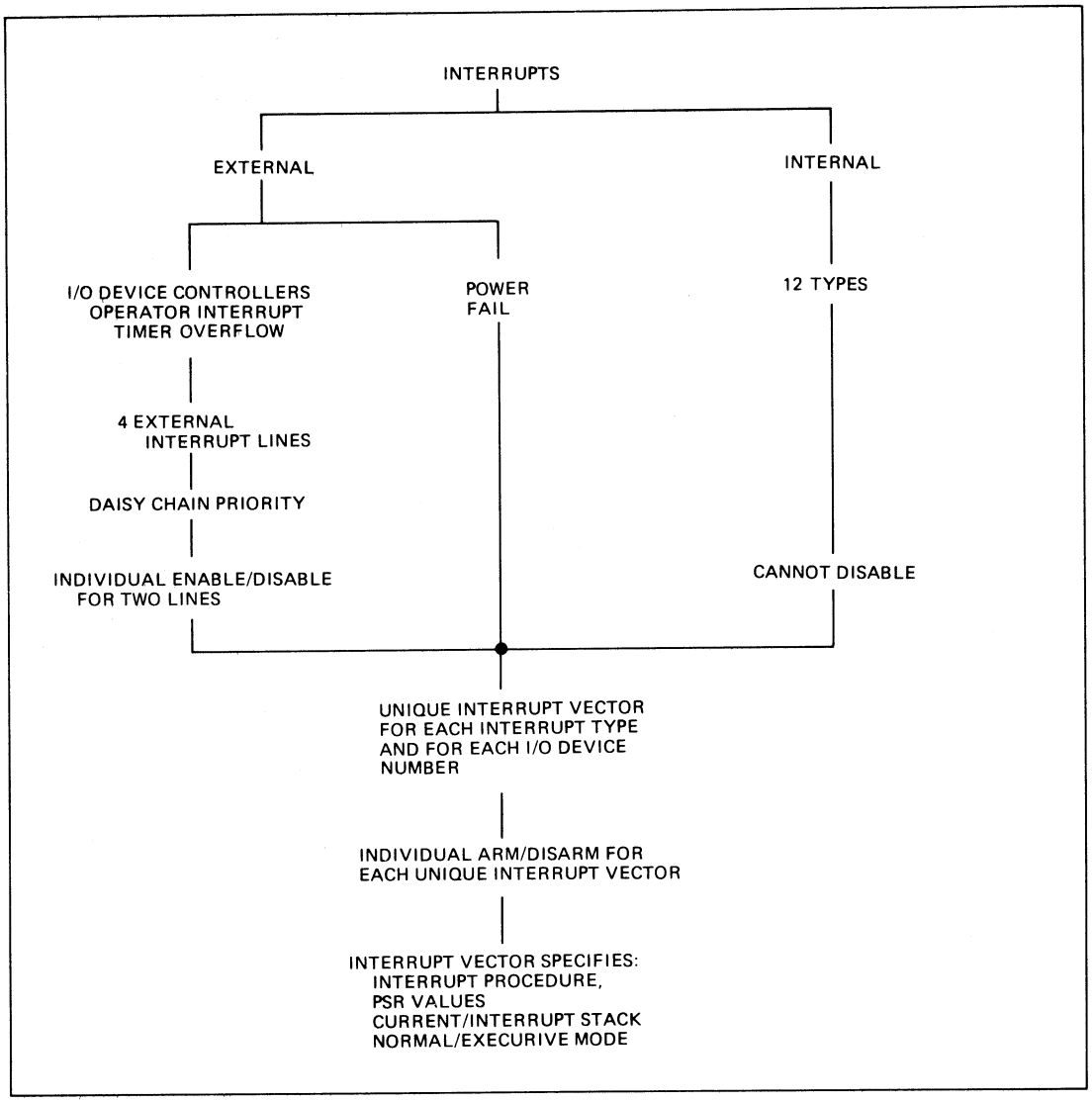


Figure A. Interrupt Architecture.

3.2 INTERRUPT VECTOR TABLE

Each type of internal interrupt, each device number associated with an external interrupt, the operator interrupt, and the timer overflow, specify a unique interrupt vector. This vector specifies whether the interrupt should be processed, and the procedure and the environment for executing the procedure.

The Interrupt Vector Table is a table of two-word entries in memory which specify the action to be taken when an interrupt is to be processed. The address of the first entry of this table is specified by a pointer in memory at location "0A". The format of the Interrupt Vector Table entry is shown in Figure A.

The Interrupt Vector Table is accessed when an enabled interrupt is honored. The address of the desired table entry is:

Interrupt Vector Table entry address = ("0A") * 4 + interrupt-vector-number * 4

The interrupt vector numbers are specified in topic 3.3. The address is the address of the first word of the desired entry.

Bit 0 of the first word of the entry is an armed bit. This bit specifies whether the particular interrupt is armed or disarmed. If it is disarmed, the interrupt is disregarded. If it is armed, the table entry specifies the interrupt processing procedure to be used and the environment to be set up by this procedure.

The definition of the fields of the Interrupt Vector Table entry are as follows:

I — Interrupt Arm — word 0, bit 0

- I = 0: interrupt is disarmed; disregard this interrupt
- I = 1: interrupt is armed; proceed with interrupt processing

X — Mode — word 0, bit 1

- X = 0: execute the interrupt procedure in the normal mode
- X = 1: execute the interrupt procedure in the executive mode

S — Stack — word 0, bit 2

- S = 0: use the currently active data stack when executing the interrupt procedure
- S = 1: use the interrupt data stack when executing the interrupt procedure

word 0, bit 3

\- — not used

MASK — Interrupt Mask — word 0, bits 7-4

This is the external interrupt line enabled/disabled mask to be placed into the Program Status Register for use when the interrupt procedure is invoked. The definition of these bits is as follows:

- bit = 0: line disabled
- bit = 1: line enabled

The bits correspond to lines as follows:

- bit 4: operator interrupt
- bit 5: timer overflow
- bit 6: external interrupt line 0
- bit 7: external interrupt line 1

PLIBN Program Library Number word 0, bits 15-8

This field is the index into the Program Library, PLIB, for the interrupt procedure to be used.

PP Program Pointer word 1

This field is the Program Pointer value to be used as the entry point into the interrupt procedure.

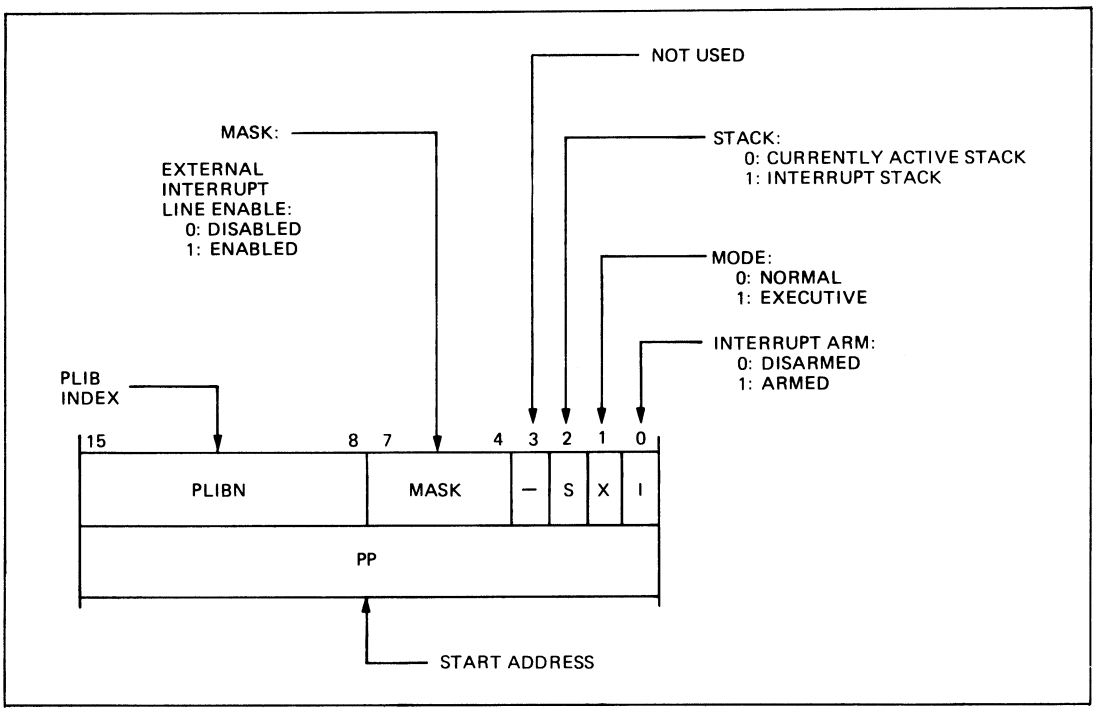


Figure A. Interrupt Vector Table Entry.

3.3 INTERRUPT DEFINITIONS

The types of internal and external interrupts are named, their interrupt vector numbers defined, and the arguments which are pushed into the top of the stack upon acknowledgment are specified.

Each type of internal and external interrupt has a unique interrupt vector number which specifies the Interrupt Vector Table entry to be used in processing it. The generation of an internal interrupt and the honoring of an external interrupt pushes an argument into the top of the stack which is active during the interrupt processing; this argument is utilized by the interrupt processing procedure. The interrupts are defined below with the interrupt vector number at the left margin and the argument at the right margin.

NOTE: Interrupt types are referred to, in this manual, by their interrupt vector number and argument: interrupt-vector-number argument.

Internal Interrupts:

1 Trace — argument: 0

An instruction of a procedure segment marked for trace has been executed.

1 Trace — argument: 1

A procedure in a segment marked for trace has been CALLed but no instructions in that procedure have been executed.

4 Supervisor Call — argument: *

A Supervisor Call instruction has been executed.

5 Program Complete — argument: 0

A Procedure Exit instruction has been executed when the latest Procedure Mark is that for a MAIN PROCEDURE.

6 Unimplemented Instruction — argument: **

An attempt has been made to execute an instruction which is not defined in the processor set.

7 Trap Instruction — argument: ***

A Trap instruction has been executed.

8 Attention Bit — argument: PLIBN

The Program Library (PLIB) descriptor retrieved in process of calling or returning to a remote program segment has its attention bit (A) set.

9 Stack Overflow — argument: 1

The stack is within 16 words of SL.

9 Stack Underflow — argument: 2

An attempt has been made to pop an empty stack.

9 SB/SL Violation — argument: 4

An attempt has been made to reference data outside the stack.

9 Privileged Instruction Violation — argument: 5

An attempt has been made to execute a privileged instruction (PNOP, XIM, MICR, or RESM) while in normal mode.

9 Privileged Addressing Mode Violation — argument: 6

An attempt has been made to execute a memory reference instruction with an absolute addressing mode (mode 7) while in normal mode.

9 Store into Program Segment Violation — argument: 7

An attempt has been made to execute a store type of memory reference instruction with a constant addressing mode (mode 6) while in normal mode.

External Interrupts

0	Power Fail	argument: **
	AC power has been lost and DC power to the processor will be available a minimum of 2 milliseconds. (DC power will be maintained to the MOS memories if an optional battery pack is attached to the machine.) See topic 13.2.	
2	Operator Interrupt	argument: **
	The interrupt button on the front panel has been depressed. Enabled/disabled by bit 4 in the PSR.	
3	Timer Overflow	argument: **
	A timer signal from the real-time clock (located in the power supply) causes the timer (addresses 4-7) to be incremented; if a carry out of the most significant bit occurs, a timer overflow interrupt will occur. When the timer interrupt is not enabled (bit 5, PSR), the counting does not occur and an interrupt cannot happen.	
15	Maximum Device Number Exceeded	argument: ****
	The number at the interrupting device exceeds the maximum device number as specified in location "000E".	
16+ Device Number	External Input/Output Interrupts	argument: ****
	An interrupt request has been honored for a device controller on one of the four external interrupt lines.	

Action Requests

In addition to the internal and external interrupts defined above, there are a number of action request conditions which are serviced by the processor. Specifically:

Parity Error: This condition arises when the processor detects a parity error when reading from memory. When the condition arises, the processor locks up. The condition is verified by a panel address display of "0051" when the CMA/FBUS is selected.

Restart See topic 13.2

Maximum Device Number Exceeded, Concurrent I/O: See topic 4.5

Load: See topic 12.6.

---

* The argument is copied from TOS of the calling stack.

** Bit 15 of the argument indicates which stack was active when the interrupt occurred:

bit 15 = 0: user stack
bit 15 = 1: interrupt stack

*** The argument is the PP for the instruction following the trap instruction.

**** Bit 15 of the argument indicates which stack was active when the interrupt occurred, and bits 9-0 indicate the interrupt source:

bits 9-0: device number
bit 15 = 0: user stack
bit 15 = 1: interrupt stack

3.4 INTERRUPT PROCESSING SEQUENCE

The sequence of steps involved in honoring and initiating the processing of an internal or external interrupt is flow charted.

Figure A flowcharts the steps followed by the 32/S processor as it recongizes an enabled interrupt request, and then sets up the environment for the interrupt procedure.

Internal interrupts are generated by execution of specific 32/S instructions. The instruction execution includes the set up of the interrupt environment.

External interrupt requests are processed between execution of 32/S instructions. If two or more interrupts are enabled at the same time the one of highest priority is honored first. The priority is:

1. Power Fail
2. Timer Overflow
3. Operator Interrupt
4. Device controller on external interrupt line 3.
5. Device controller on external interrupt line 2.
6. Device controller on external interrupt line 1.
7. Device controller on external interrupt line 0.

The power fail interrupt and external interrupt lines 2 and 3 are always armed. Timer overflow, operator interrupt and external interrupt lines 0 and 1 may be armed or disarmed. Interrupts are only honored for armed interrupt sources.

Any number of I/O device controllers attached to a given external interrupt line may desire to signal an interrupt request simultaneously. A hardware daisy chain priority network assigns priority to the I/O device controllers on the basis of their position within the chassis; the priority goes from high to low, moving from the foremost card slot to the rear of the chassis. Only the highest priority controller desiring to interrupt is permitted to present its interrupt request. After this interrupt has been honored, the next highest priority controller desiring to interrupt presents its interrupt request.

A "maximum device number exceeded" interrupt request may be generated during the processing of an interrupt requested by a device controller. If this occurs, the maximum device number exceeded interrupt is taken rather than the external I/O interrupt requested by the device controller.

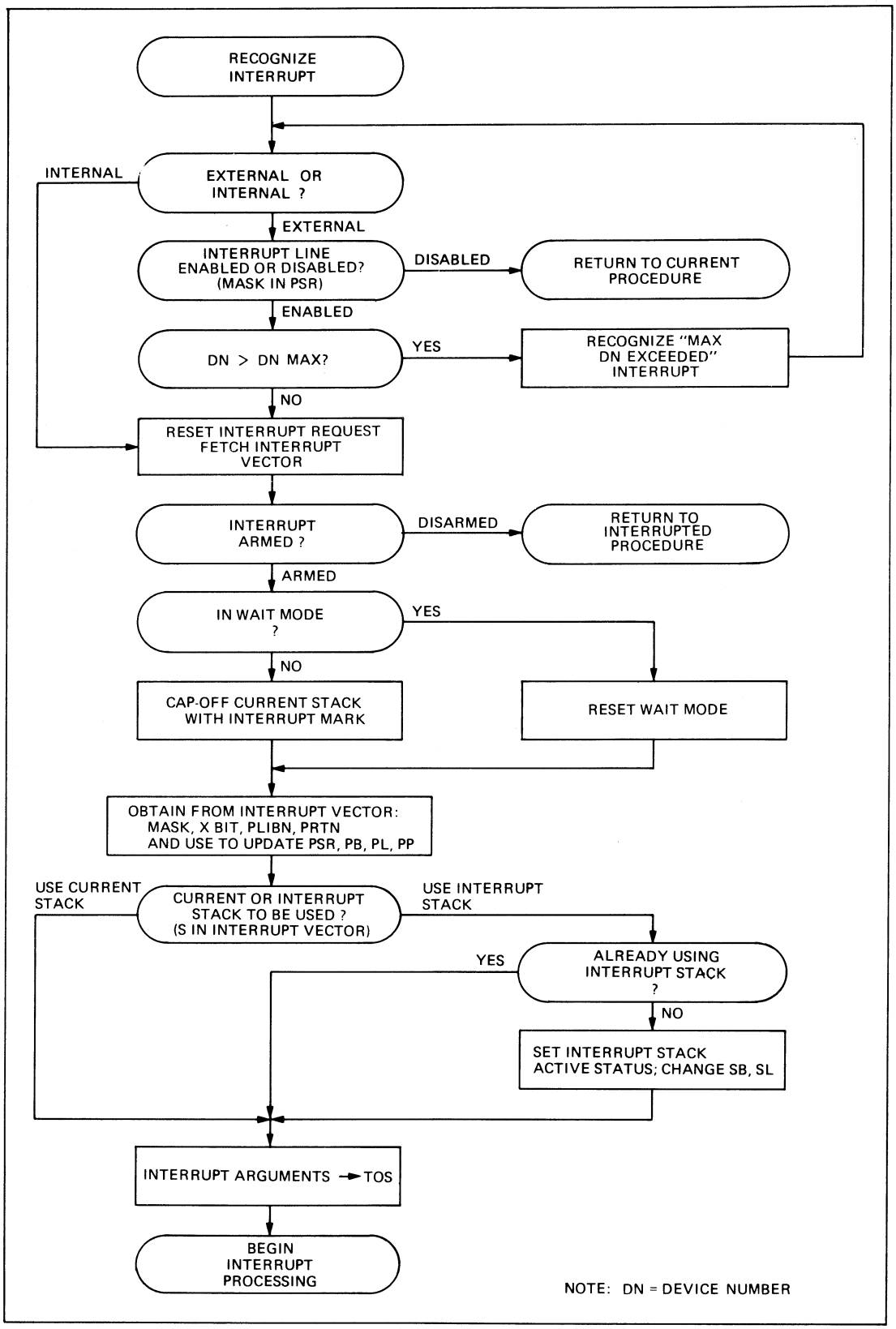


Figure A. Interrupt Processing.

3.5 DATA STACK AS PROCESS INTERRUPT

When an interrupt is serviced within the currently active stack the environment for the interrupt interrupt procedure is created by the processor and rolled back by an IXIT instruction.

Figure A shows a sequence of four snapshots of the data stack as an interrupt is processed for the case in which the Interrupt Vector Table entry specifies that the interrupt is to be processed in the currently active stack.

The occurrance of an interrupt causes the contents of the active stack head registers to be pushed into the stack in memory, creates the Interrupt Mark, and pushes the interrupt argument into the stack. The interrupt procedure then executes in the environment above this Mark. An IXIT (Interrupt Procedure Exit) instruction (see topic 9.5) terminates the interrupt procedure by rolling back the environment created by the Interrupt Mark.

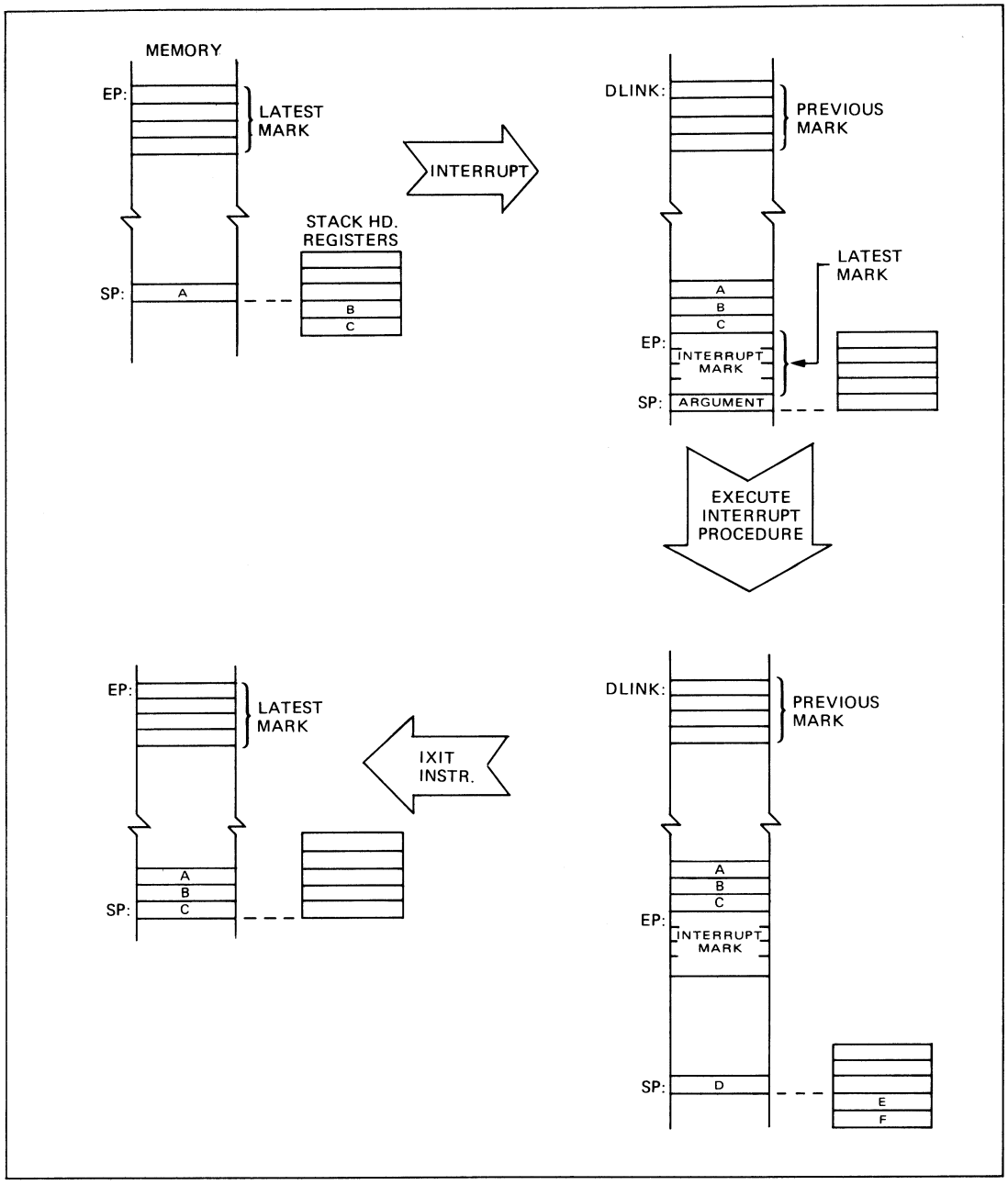


Figure A. Data Stack as Process Interrupt in Current Stack.

3.6 DATA STACK AS PROCESS INTERRUPT (Continued)

When the servicing of an interrupt requires changing from a user stack to the interrupt stack, the processor caps off the current user stack and activates the interrupt stack.

Figure A shows a sequence of snapshots of the current user stack and of the interrupt stack as an interrupt is processed, for the case in which the Interrupt Vector Table entry specifies that stacks be switched. Four pictures of the user stack are shown across the top of the figure, and the four corresponding pictures of the interrupt stack are shown across the bottom of the figure.

To describe the sequence of events associated with switching stacks to service an interrupt, it is helpful to first describe how a software executive (operating system) might control the activity of the stacks. The executive itself uses the interrupt stack as its own data stack. Within this stack it maintains an Active Process Table which contains the Stack Base and Stack Length parameters for each user stack.

The system is activated with the executive running and with the interrupt stack as the active stack. The user stacks are all capped off with Interrupt Marks and inactive. When the executive wishes to activate a user it places a Stack Descriptor word, SD, into the top of the stack and executes a RESM (Resume) instruction. This instruction uses the SD to access the Stack Base and Stack Length parameters for the desired user stack and activates this stack. This RESM instruction also caps off the interrupt stack with an Interrupt Mark and inactivates it.

When an interrupt occurs and the Interrupt Vector Table entry for that interrupt specifies switching to the interrupt stack, this entry will also specify that control be returned to the executive. The processor then inactivates the user stack and activates the interrupt stack. An interrupt procedure within the executive then processes the interrupt. The interrupt procedure is terminated by an IXIT (Interrupt Procedure Exit) instruction (see topic 9.5). This rolls back the interrupt stack to the environment which existed before it activated the user.

Subsequently, the executive will determine which user to activate next. It then pushes the desired Stack Descriptor into its stack and executes a RESM instruction.

The stack drawings in Figure A show a sequence which begins at the point when a user is active (leftmost user and interrupt stacks). When the interrupt is acknowledged, the interrupt firmware pushes the active stack head registers into the user stack in memory. The processor then caps off the user stack with an Interrupt Mark and inactivates it. Finally, the processor activates the interrupt stack and pushes the interrupt argument into this stack (the argument is not shown). The interrupt procedure is then processed in the interrupt stack. (See second-from-left user and interrupt stacks.)

At the termination of the interrupt procedure an IXIT instruction is executed. This rolls back the interrupt stack to the environment preceding its Interrupt Mark. (This Interrupt Mark had been created when the executive originally executed a RESM to activate the user.) The interrupt stack remains active as the executive performs other tasks.

At the point when the executive desires to activate a user, it executes a LADR (Load Address) instruction to push the Stack Descriptor, SD, into its stack. The status of the user and interrupt stacks are then as shown in the second from right column of pictures.

The executive then executes a RESM instruction. This instruction activates the user stack specified via the SD pointer to the Active Process Table. It caps off the interrupt stack with an Interrupt Mark and makes it inactive. (Rightmost user and interrupt stacks.)

Note that, for simplicity, the figure shows a single user stack being interrupted, inactivated, and then reactivated. However, in a multi-user stack environment the executive would presumably activate a new user after servicing the interrupt.

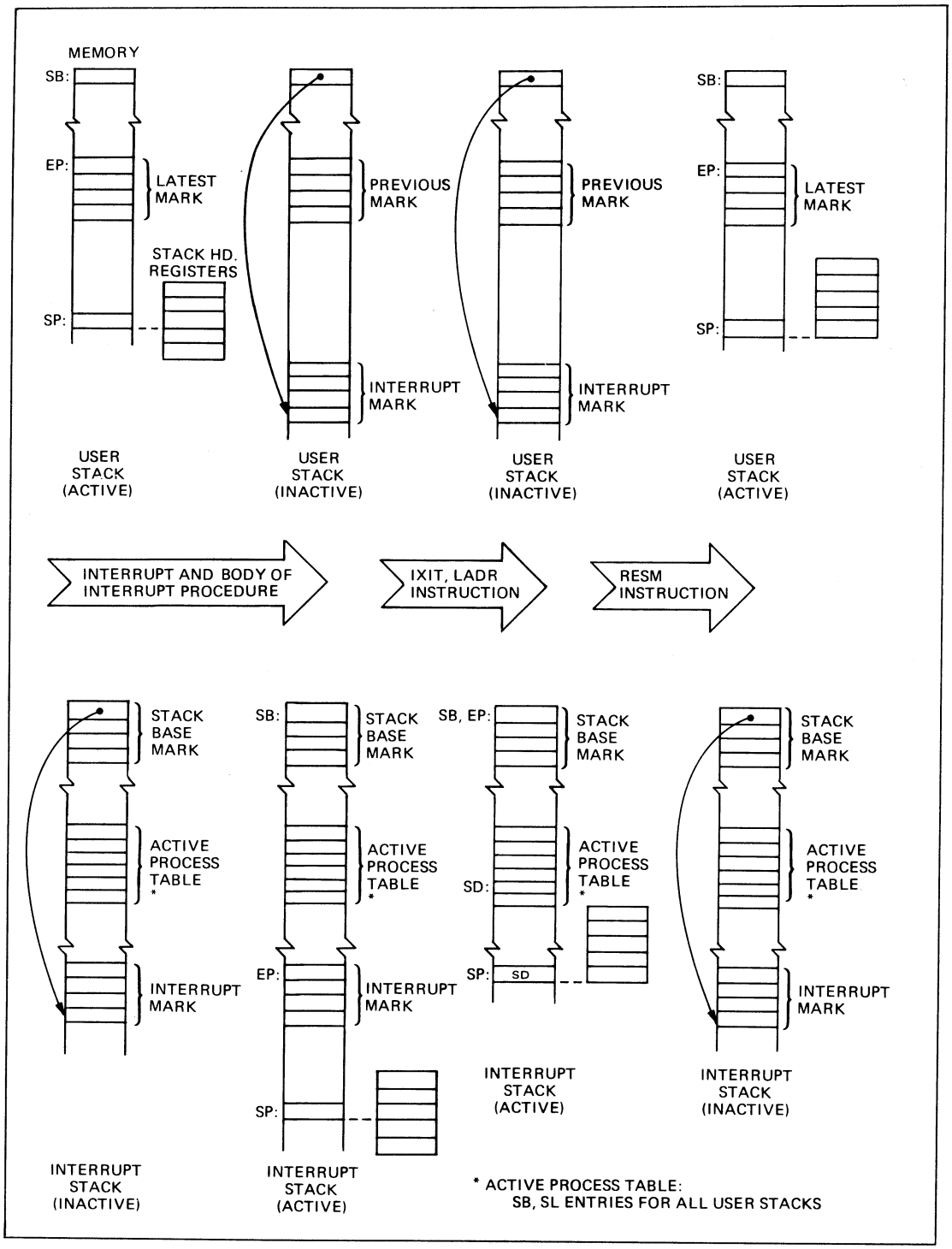


Figure A. User and Interrupt Stacks as Process an Interrupt.

4 Input/Output

4.1 TYPES OF I/O

The 32/S provides three types of I/O facilities, each providing the optimum hardware cost/programming/speed tradeoff for a particular I/O situation.

Three types of I/O facilities are provided in the 32/S computer: programmed I/O, concurrent I/O, and direct memory access I/O. (See Figure A.) Programmed I/O permits the transfer of a single byte or word. Concurrent I/O and direct memory access I/O accomplish the transfer of a block of data. The concurrent I/O facility is used for low speed peripheral devices, and the direct memory access I/O is used for high-speed peripheral devices.

A programmed I/O operation is accomplished by executing a load instruction or a store instruction from or to a Monobus location which is assigned to the desired register within the desired peripheral device controller. The controllers contain registers for status, mode, order (command) and data, and for device parameters (such as sector and track addresses in a disc controller). The group of registers for a particular controller are assigned to a block of Monobus locations which are referred to as a Device Register Block, DRB. The format of the DRB used in standard Microdata controllers is discussed in topic 4.2.

A concurrent I/O operation is initialized by programmed I/O operations, but the data block transfer is then controlled by the processor without program intervention. Thus the program may perform other operations while data transfers are occurring. The transfer of the individual byte or word is accomplished between execution of 32/S instructions. The controller generates a concurrent I/O request whenever it is ready to transfer the byte or the word. To transfer the data, the processor refers to the Concurrent I/O Control Block (in memory) for the requesting device controller to determine the memory address for the next byte or word to be transferred. Information in this block determines if the block transfer is to be terminated. (The block is initialized by software before the transfer operation is triggered.) The processor commands the controller to stop requesting I/O when the block transfer is complete. The controller may request an external interrupt when informed that the transfer is terminated. (See topic 4.5 and 4.6.)

A direct memory access I/O operation is also initialized by programmed I/O operations, but the data block transfer is then controlled by hardware within the device controller instead of information contained in a CIOCB. The transfer of the individual byte or word is accomplished whenever the controller can get priority to use the Monobus. The controller hardware determines when the transfer is to be terminated, and may request an external interrupt at that time.

The sequence of I/O operations is shown in Figure B. The solid lines of the flowchart indicate the sequence followed by the 32/S processor. The dotted lines indicate that the direct memory access I/O operates in the background, and does not involve the processor.

Programmed I/O may be used to effect data transfers on a single byte or word basis with low speed devices such as a TTY, CRT, card reader, line printer, paper tape punch or paper tape reader. Concurrent I/O may be used to effect data block transfers with these same types of devices. Direct memory access I/O is used with high speed devices such as a magnetic tape unit or disc drives.

PROGRAMMED I/O	SINGLE BYTE/WORD TRANSFER ● DATA ● STATUS ● COMMAND	● DATA TRANSFER EXECUTED BY LOAD AND STORE INSTRUCTIONS
CONCURRENT I/O	BLOCK TRANSFER OF DATA FOR LOW-SPEED DEVICES	● INITIALIZED BY PROGRAMMED I/O ● DATA TRANSFER ● EXECUTED BY PROCESSOR ● DATA TRANSFER BETWEEN INSTRUCTION EXECUTIONS
DIRECT MEMORY ACCESS I/O	BLOCK TRANSFER OF DATA FOR HIGH-SPEED DEVICES	● INITIALIZED BY PROGRAMMED I/O ● DATA TRANSFER EXECUTED BY CONTROLLER HARDWARE ● DATA TRANSFER AS SOON AS CONTROLLER GETS PRIORITY TO USE MONOBUS

Figure A. Types of I/O.

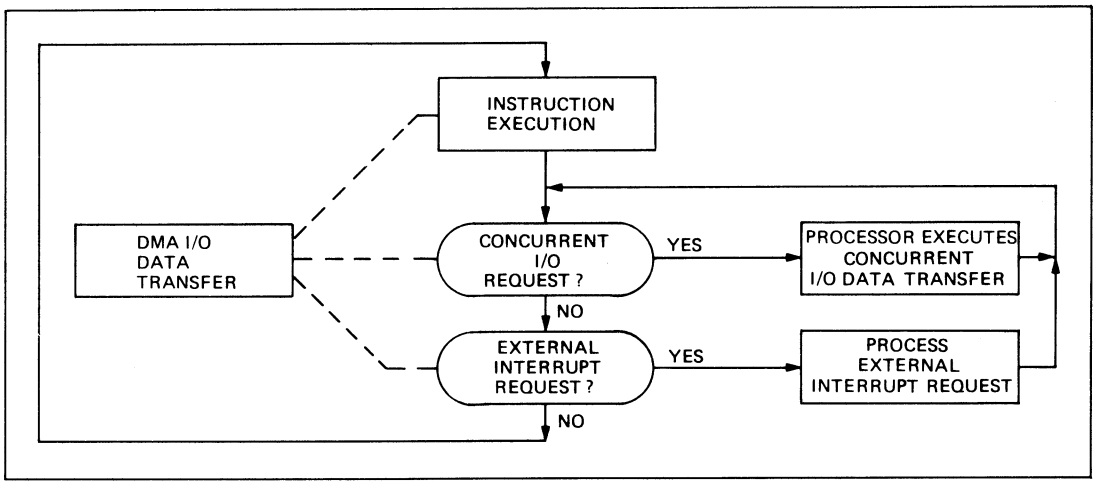


Figure B. Sequence of I/O Operations.

4.2 DEVICE REGISTER BLOCK

Each device controller has an eight-word block of Monobus locations called a Device Register Block, DRB, through which its registers can be accessed. Each interrupt line has a single word location through which the device address of an interrupting controller can be accessed.

An eight-word block of Monobus locations is reserved for each I/O device controller number. These locations correspond to the status, order, data and other registers within the device controller module. They are read and written as though they were locations within main memory. The block of locations is referred to as a Device Register Block, DRB.

The DRB's begin at Monobus word location "3C000". Each DRB begins at the location "3C000" + 16 * (device number). A maximum of 1024 DRB's is available, although the last one (starting at "3FFF0") is assigned to a special role associated with interrupts and with the front panel. See Figure A.

A device controller module may contain more than one device controller, and therefore more than one device number and DRB associated with it. For example, the Multi-Purpose I/O controller module, which interfaces to several peripherals of different types, has four DRB's. On the other hand, the disc controller module, which interfaces with up to four disc drives, has a single DRB. Switches on the controller modules are used to assign device numbers and to thereby assign DRB locations.

The standard format of the eight-word DRB is shown in Figure A. The status word contains bits which indicate the ready status of the peripheral device, interrupt requests which are pending, and error conditions. The order byte is used to specify an order, or command, to the controller. The mode byte is used to set up to four different types of modes, such as the interrupt mode, parity mode, etc. The data word (or the least significant byte of this word) is the input data or output data for the device. The extended status and extended order words provide optional extensions to the status word and order byte field.

The word labeled disc address is used to specify platter, head, track, and sector addresses in the disc controller. The last two words of the DRB are the data buffer bookkeeping registers used in controllers with direct memory access I/O capabilities. The next DMA data address is initialized to point to the low-address end of the data buffer. The remaining DMA byte count is initialized to the byte count of the data buffer. As each data word (or byte) is transferred, the controller hardware increments the next DMA data address and decrements the remaining DMA byte count.

Standard formats for the status word, order byte, and mode byte are presented in separate topics within this section. The specific formats of each field of the DRB for particular device controllers are given in the I/O appendix to this manual.

The highest-address DRB is assigned to a special purpose. The first four word locations contain the device number of the device requesting an interrupt on each of the four external interrupt lines. The fifth word location contains the device number of a device requesting a concurrent I/O transfer. See topic 4.3 for a further explanation.

The last four locations of the highest address DRB are associated with the maintenance front panel. (See topic 4.12.)

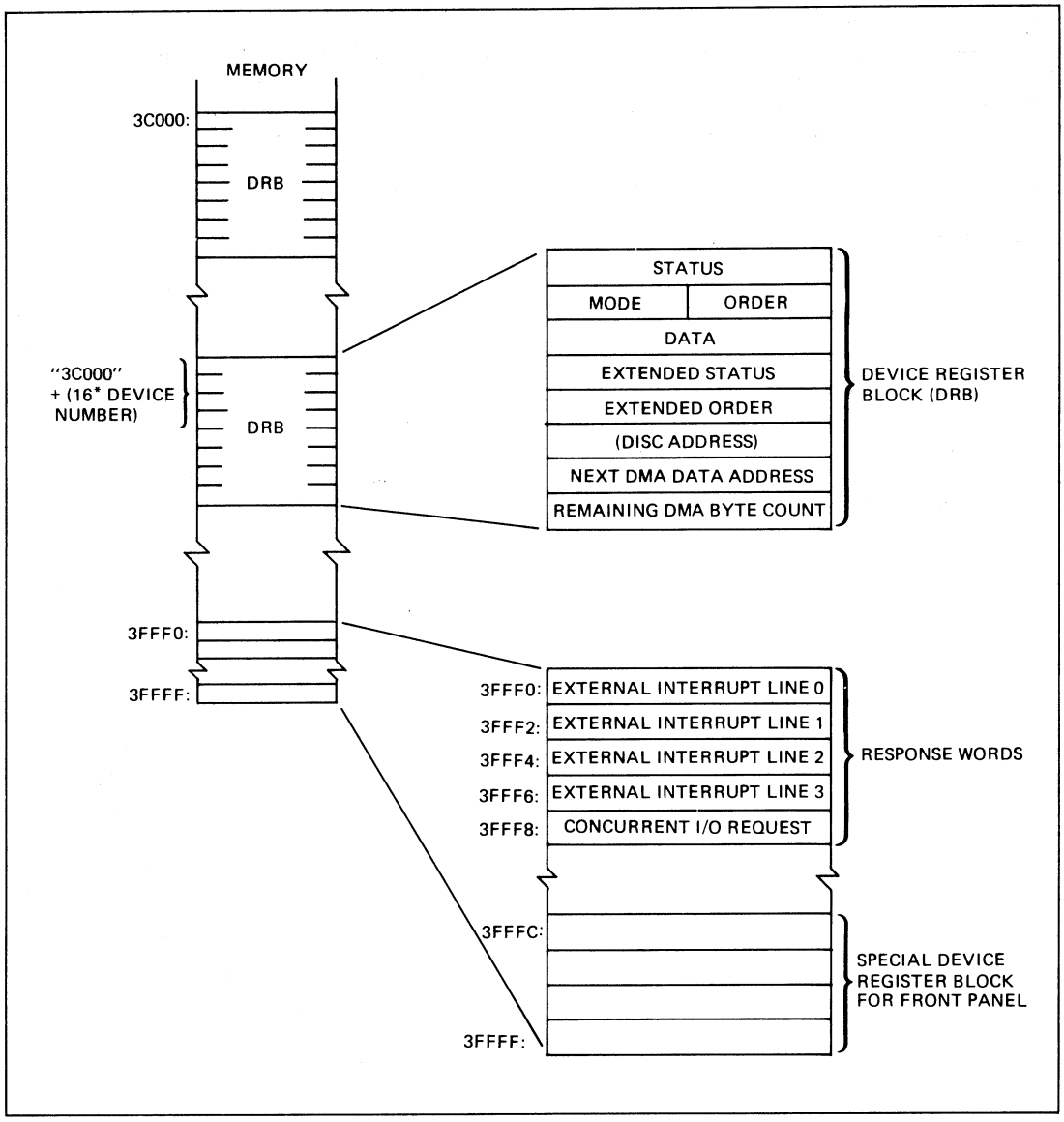


Figure A. I/O Monobus Map.

4.3 CONTROLLER RESPONSE WORD

Device Controllers provide a response word to specify their device number and other information when they request an external interrupt or a concurrent I/O transfer.

Each device controller is designed to provide a response word when its request for an external interrupt, or for a concurrent I/O data transfer, is honored. This response word identifies the interrupting controller to the processor. The sequence of operations involved in accessing the response word is discussed in topic 4.4.

The format of the response word is shown in Figure A. Bits 13 through 4 contain the controller's device number. Bits 0 and 1 are utilized for concurrent I/O transfer requests to inform the processor of the type of data transfer to be performed. Specifically:

bit 0 = 0:	input data transfer
bit 0 = 1:	output data transfer
bit 1 = 0:	transfer word
bit 1 = 1:	transfer byte
bits 3, 2:	zeroes
bits 13-4:	device number
bits 15, 14:	zeroes

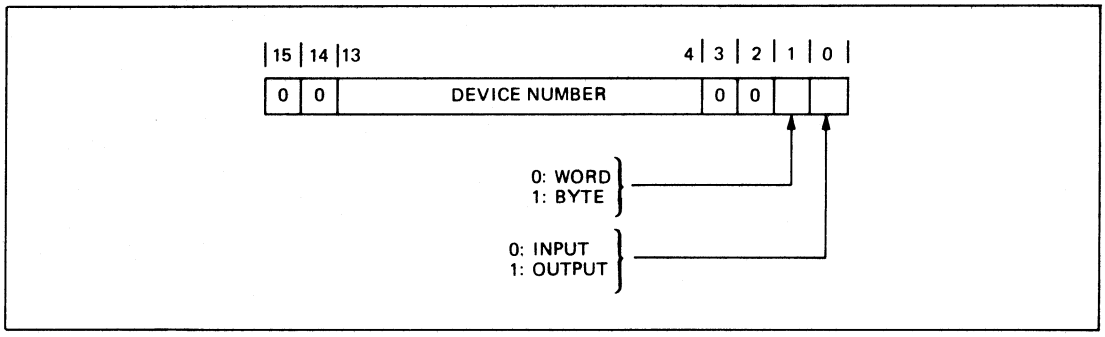


Figure A. Response Word.

4.4 CONTROLLER INTERRUPT OPERATION

The 32/S computer provides four external interrupt lines. Any number of controllers may be set to request interrupts on any one of the four lines. Interrupt requests on each line are acknowledged in the priority established by the physical placement of the controllers in the chassis.

The 32/S computer provides four external interrupt lines. Controllers may be manually set to request interrupt on any one of these four lines. Two lines are independently enabled or disabled by the mask in the Program Status Register; the other two lines are always enabled. (See topic 2.16.) (In addition, each controller's interrupts may be individually enabled or disabled by setting its interrupt mode. See topic 4.9.)

The processor honors requests on the enabled priority lines in the priority order of line 3 through line 0. The controllers respond according to a daisy-chain priority which is established by their position within the chassis, with the highest priority position being the front-most card slot. See Figure A.

When the processor determines that there is an external interrupt request on a particular enabled external interrupt line it reads the Monobus location of the corresponding response word. The controller with the highest priority, according to the priority daisy chain, which is requesting an interrupt on that line, responds by outputting its response word on to the Monobus.

The external interrupt line assignment of a device controller specifies which one of four external interrupt response word addresses that controller can respond to. Specifically:

External Interrupt Line Assignment	Response Word Monobus Address
0	"3FFF0"
1	"3FFF2"
2	"3FFF4"
3	"3FFF6"

(See topic 4.2.)

The interrupt process is flow charted in Figure B. When an interrupt condition arises in a controller, the controller sets a bit to indicate this condition in the status word of its DRB. No further action occurs unless or until the controller is placed into the enable interrupts mode (by writing into the mode byte of the DRB).

If the controller is in the enable interrupt mode it signals its assigned external interrupt line. When the processor responds to the request on this interrupt line it reads the corresponding response word location. If the controller is the highest priority unit which is signaling on this external interrupt line it responds by outputting its response word. It then releases the external interrupt line.

The processor utilizes the device number field of the response word to index to an entry in the Interrupt Vector Table. (See topic 3.2.) A bit in this entry indicates whether the interrupt for the requesting controller is armed or disarmed. If the interrupt is disarmed, the interrupt request is disregarded. If the interrupt is armed, the interrupt procedure specified in the Interrupt Vector Table entry is invoked.

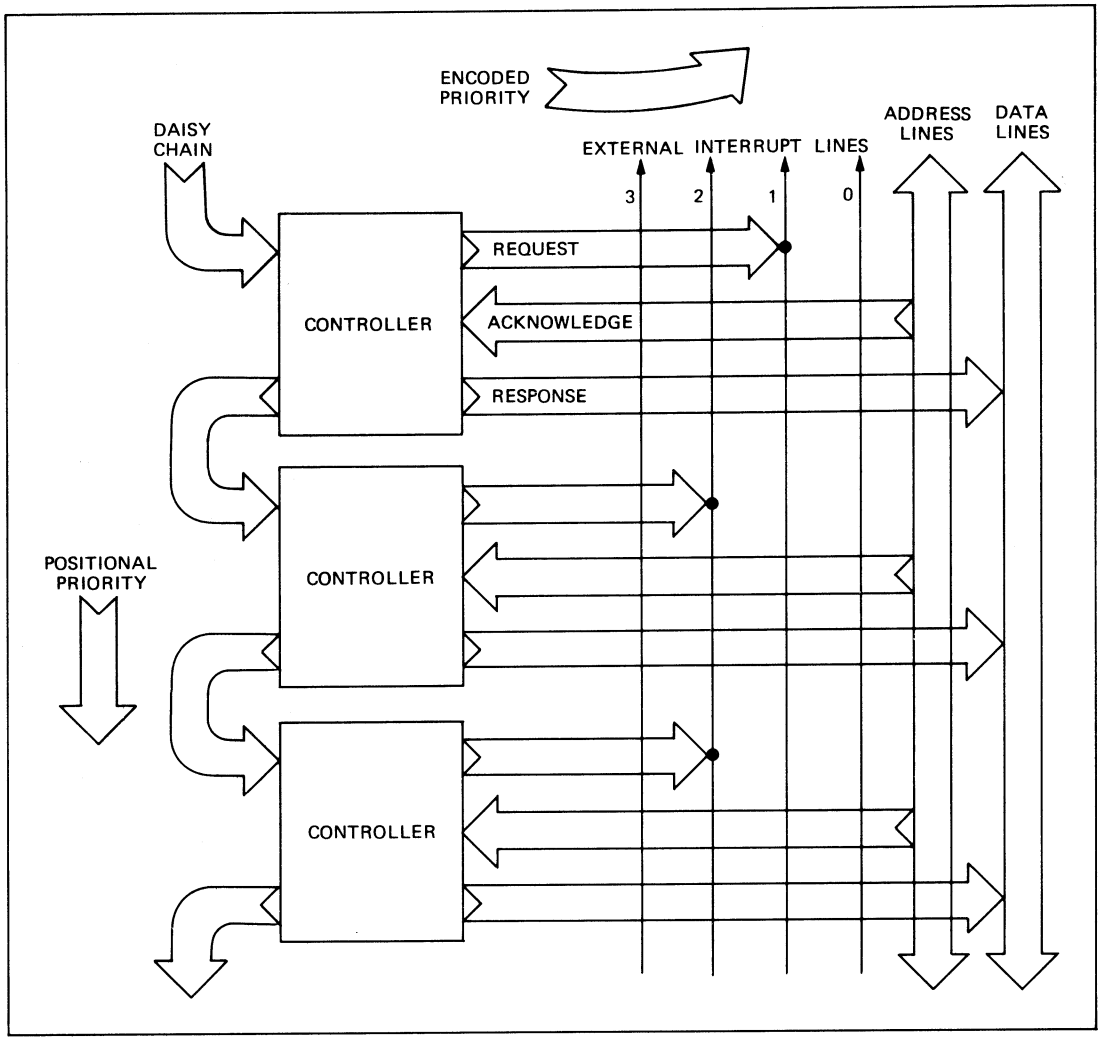


Figure A. Interrupt Lines, Block Diagram.

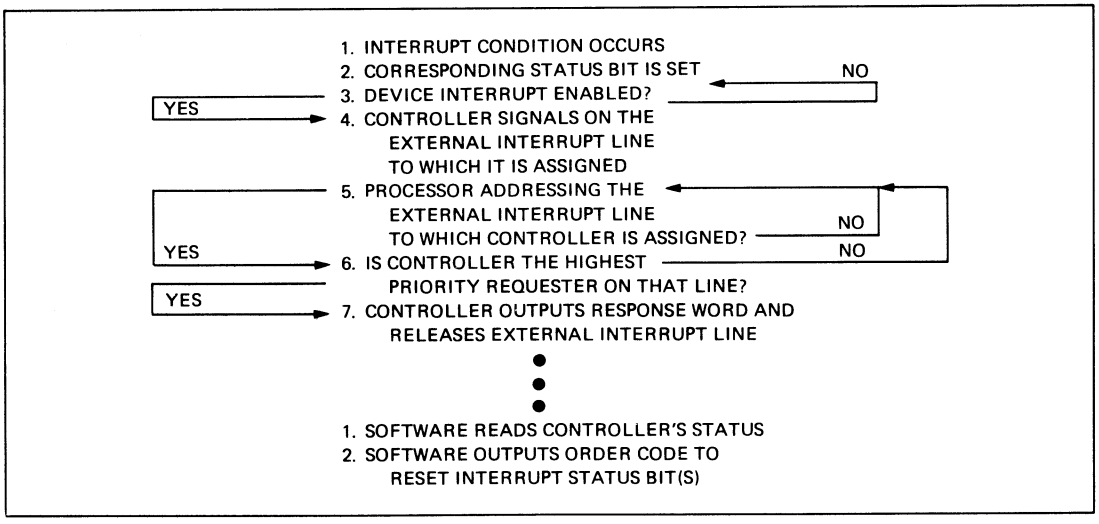


Figure B. Flowchart for Controller Interrupt Sequence.

4.5 CONCURRENT I/O CONTROL BLOCK

The processor's concurrent I/O facility provides the control of block transfer I/O operations utilized by lower speed devices. The block transfer may be terminated either on the basis of data byte count or data byte match.

The processor provides the control logic for controlling data block transfers to and from device controllers designed for concurrent I/O operation. Concurrent I/O utilizes a four-word control block in main memory for each device number to keep track of the current data buffer address and to determine if the block transfer is to be terminated. The block transfer termination may be specified either on the basis of byte count or byte match. The actual data transfer occurs when the processor acknowledges a concurrent I/O request from the controller.

The location in memory of the Concurrent I/O Control Block, CCIOB, is specified by a pointer in location "00008" and by the device number. (See Figure A.) Specifically:

$$\text{address of first word of Concurrent I/O Control Block} = \left(\text{("00008")} + 2 * \text{device number}\right) * 4$$

The format of the Concurrent I/O Control Block is shown in Figure A.

The second word of the block and the AA field of the first word specify the address of the next data word or byte to be transferred. Specifically:

$$\text{address of next word or byte to be transferred} = (\text{AA}) * 2^{16} + \text{(next-data-address-word)}$$

This address is initialized by the software to the lowest address of the data buffer.

The third word of the block specifies the byte count remaining to be transferred. The software initializes this word to the number of bytes to be transferred.

The fourth word of the block is used in specifying a block transfer on the basis of a byte match. Two criteria of byte match are provided: a match on the basis of a compare, and a match on the basis of a bit table. The type of match is specified by the T bit in the first word of the block. Specifically:

T = 0: match on basis of compare

T = 1: match on basis of bit table

For a match on the basis of the bit table, each data byte transferred is used as an index to a bit table. If the bit table contains a 1 bit at the indexed position, the match is successful. If the table contains a 0 bit, the match is unsuccessful. For example, if the byte being moved has the value "0F" then the 16th bit of the table is tested. If the 16th bit is a 1, the match is successful and data transmission is terminated.

The address of the 256-bit table is specified by the fourth word of the block and the BB field of the first word of the block. Specifically:

$$\text{address of first word of bit table} = (\text{BB}) * 2^{16} + \text{(byte-match-control-word)}$$

For a match on the basis of compare, the fourth word of the Concurrent I/O Control Block provides an offset byte and a compare byte which are the matching criteria. The match comparison process is as follows: the data byte transferred is added to the most significant byte, and the result is masked to seven bits (modulo 128). This "adjusted data byte" is then compared to the least significant byte. If the adjusted data byte is less than the least significant byte, the match is successful. Specifically:

$$\left\{\left[\text{byte-match-control (bits 15-8)}\right] + \text{data}\right\} \text{ AND "7F"} < \left[\text{byte-match-control (bits 7-0)}\right] : \text{match}$$

The basis for the block transfer termination is specified by the S bit in the first word of the block. Specifically:

S = 0:	terminate on basis of byte count exhausted
S = 1:	terminate on basis of byte match or on the basis of byte count exhausted, whichever occurs first.

The byte count is considered to be exhausted when it has been:

byte data:	decremented to 0 or -1
word data:	decremented to 1, 0, -1, or -2

NOTE: The device number supplied by the controller is compared against the number of entries in the Concurrent I/O control block (location "10", see topic 2.15). If the device number exceeds this maximum value, the processor locks up. The condition is verified by a panel address display of 3080 when the CMA/FBUS is selected.

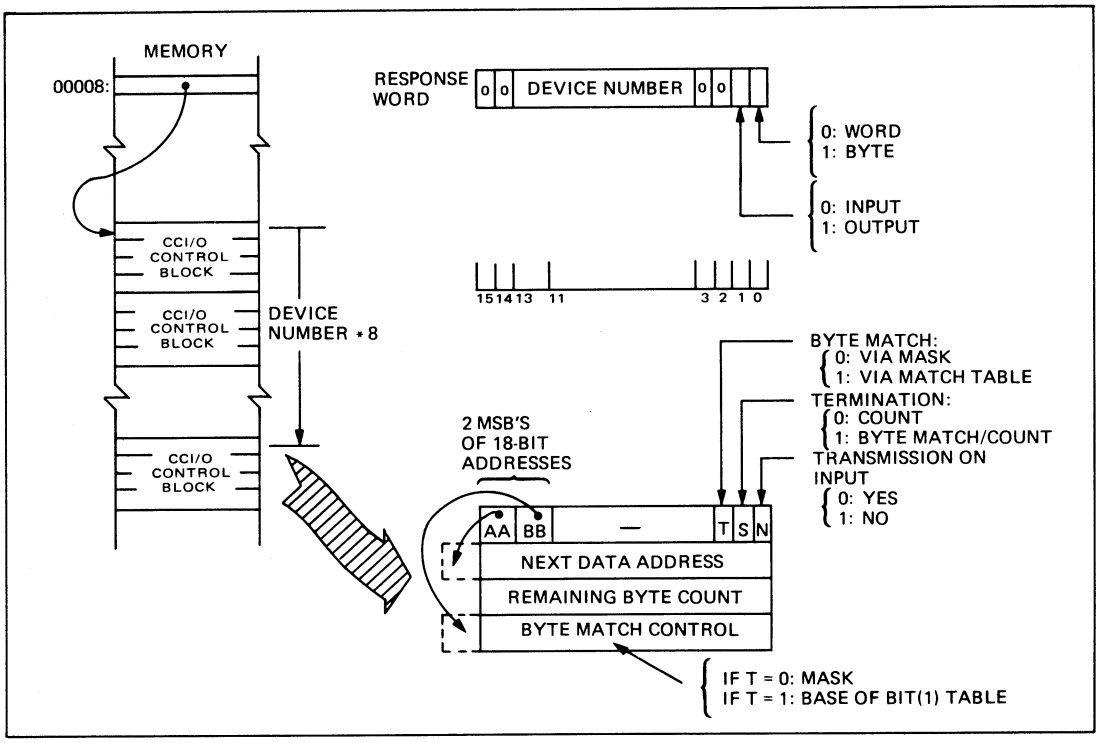


Figure A. Concurrent I/O Control Block.

4.6 CONCURRENT I/O SEQUENCE

A concurrent I/O transfer is initiated by outputting a start device (in concurrent I/O) order. The processor then performs data transfers in response to requests from the controller.

A concurrent I/O data transfer is initiated by the software storing a "start device in concurrent I/O" order to the device controller. Previous to this, the software has set up the data buffer area in main memory and initialized the Concurrent I/O Control Block for the desired device.

When the device controller is ready to transfer a data byte or word, it signals on the concurrent I/O request line. The 32/S processor tests this line after execution of each 32/S instruction. The processor acknowledges the request by reading the concurrent I/O request response word location (address "3FFF8", see topic 4.2). The controller with the highest priority, according to the priority daisy-chain, which is requesting a concurrent I/O transfer, responds by outputting its response word onto the Monobus.

When the controller is ready to transfer data, it signals the processor. After completion of the current instruction the processor determines the address of the Concurrent I/O Control Block from the device address field of the response word. It also determines, from bits in the response word, whether it is to do a word or byte transfer, and whether the transfer is input or output. The processor then performs the data transfer by reading or writing the data byte/word of the controller's DRB, decrements the remaining byte count word of the Concurrent I/O Control Block, and then determines if the transfer is to be terminated. (See Figure A.)

After performing the data transfer, the processor resumes instruction execution.

If a terminate condition exists, the processor writes a stop device order into the controller's DRB. This will generate a terminate interrupt condition in the controller which will result in an external interrupt request (if the controller is in the interrupt enable mode).

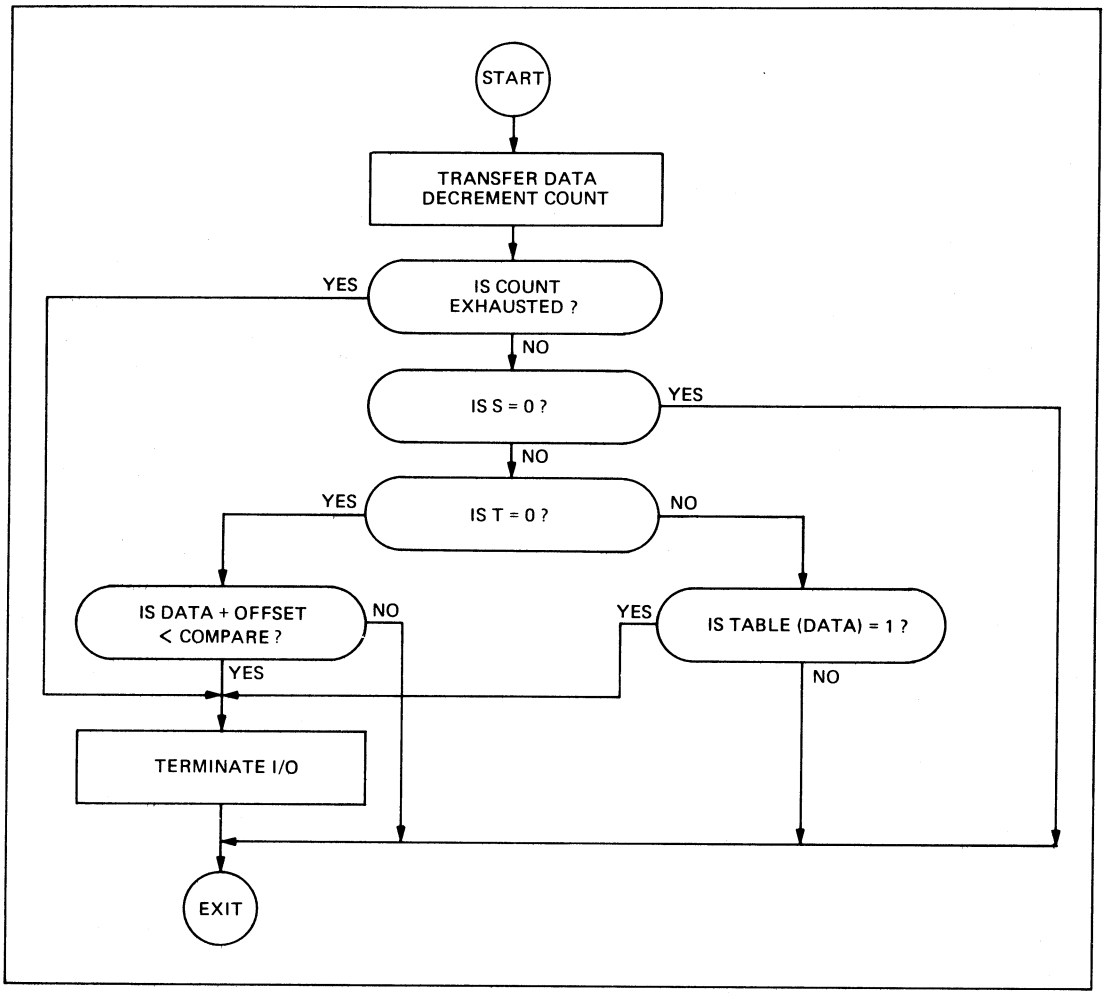


Figure A. Concurrent I/O Flowchart.

4.7 STATUS WORD FORMAT, DEVICE REGISTER BLOCK

The standard format of the DRB's status word provides four bits of ready status, four interrupt pending bits, and eight bits of general status.

The standard format of the status word of the Device Register Block is shown in Figure A. Not all controllers will actually use all of the bits defined in the standard format. The status word of the DRB is a read-only Monobus word; its contents are modified only by the logic within the controller. The definition of these bits is as follows:

bit 0 Controller Busy

This bit is a 1 if the controller has been given a start order and has not yet terminated activity.

bit 1 Device Ready

This bit is a 1 if the peripheral device is available to transfer data.

bit 2 Data Service

This bit is a 1 if the controller has data ready for input to the computer, or if it is ready to accept the next data from the computer.

bit 3 Device Writable

This bit is a 1 if the peripheral device is an output device (e.g., line printer) and if the device is not in a write-protected mode.

bit 4 Data Service Interrupt Pending

This bit becomes a 1 when the controller requests a programmed I/O data transfer (bit 2 goes from 0 to 1) This condition can only occur when the device is started in programmed I/O mode. The bit is reset by issuing a reset interrupt order.

bit 5 Terminate Interrupt Pending

This bit becomes a 1 when the controller goes from the busy state to the not busy state. This termination may be a result of issuance of a stop order to the controller or end of record condition (e.g., 80th column read from a card), or of an error condition which occurred in a controller operating in the stop on error mode. The termination can also occur as the result of starting a not ready device, or by a busy device becoming not ready. The bit is reset by issuing a reset interrupt order.

bit 6 Ready Change Interrupt Pending

This bit becomes a 1 whenever the peripheral device changes its ready state. The bit is reset by issuing a reset interrupt order.

bit 7 Special Interrupt Pending

This bit becomes a 1 when a set special interrupt order is issued to the controller. The bit is reset by issuing a reset interrupt order.

bit 8 Error Bit(s) Set

This bit is a 1 if one or more error conditions are detected by the controller.

bit 9 Data Overrun

This bit becomes a 1 if the controller has lost data because the computer has not moved previous data soon enough. The bit is reset on a start command.

bit 10 Device Parity Error

This bit becomes a 1 if the controller detects a parity error in data input from the peripheral device. The bit is reset on a start command. The controller must be operating in a check parity mode for this bit to be set.

bit 11 Bus Parity Error

This bit becomes a 1 if a parity error occurs when a controller reads data from memory. This feature may be implemented in a direct memory access type of controller, and requires that the parity option be installed in the processor and in the memory modules. (The parity bits are actually generated and checked by logic within the processor; the processor provides the parity check result to controllers on the Monobus.) The status bit is reset on a start command.

bit 12 Spare Bit No. 1

This bit is available for implementing device-peculiar types of status (e.g., motion check error on a card reader).

bit 13 Spare Bit No. 2

This bit is available for implementing device-peculiar types of status.

bit 14 Alarm

This bit is set to a 1 when the controller determines that an alarm status exists within the peripheral device (e.g., hopper check on a card reader). The bit is reset when the alarm condition is eliminated in the device.

bit 15 Operation Abort

This bit is set to a 1 when the controller determines that it must abort its current operation. Whenever this bit is set, terminate is also set. It is reset when a new order is issued.

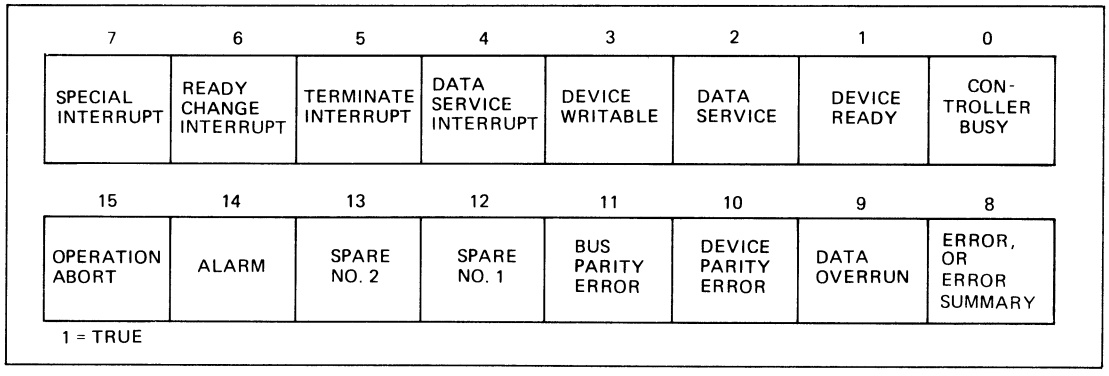


Figure A. Status Field of Device Register Block.

4.8 ORDER BYTE FORMAT, DEVICE REGISTER BLOCK

The standard format of the DRB's order byte consists of a four-bit order code and four bits which specify specific commands.

The standard formats used in writing the order byte of the Device Register Block are shown in Figure A. Seven standard four-bit order codes are defined. For some of these order codes the remaining four-bits specify specific command information. The format of the byte obtained in reading the DRB order byte contains a zero order code in the least significant four bits; the definition of the most significant four bits is device-dependent.

NO OPERATION — Order Code: "0"

This order causes no operation within the controller.

START DEVICE — Order Code: "2"

This order starts the device performing a data transfer operation.

The standard definition of the specific command bits for concurrent I/O devices is as follows:

bit 4 = 0: begin programmed I/O data transfer.
bit 4 = 1: begin concurrent I/O block data transfer.

bit 5 = 0: perform an input operation.
bit 5 = 1: perform an output operation.

bit 6 = 0: continue to run the I/O transfer even if an error occurs.
bit 6 = 1: stop the I/O transfer if an error occurs.

bit 7 = 0: operate device in forward motion direction.
bit 7 = 1: operate device in reverse motion direction.

INITIAL PROGRAM LOAD — Order Code: "C"

This order code starts the device performing an Initial Program Load, IPL. The controller inputs an IPL record into memory, starting at location 0, and then stops the transfer and causes a power restart. (See topic 12.6)

STOP DEVICE — Order Code: "4"

This order code stops the device.

DEVICE CONTROL — Order Code: "6"

This order code specifies a device-dependent command to the controller. The most significant four bits are interpreted by the controller.

SET SPECIAL INTERRUPT — Order Code: "8"

This order code sets the Special Interrupt Pending status bit (bit 7) of the DRB's status word.

RESET INTERRUPT Order Code: "A"

This order code resets the interrupt pending bits in the DRB's status word. The definition of the specific command bits are as follows:

bit 4 = 0: no action.
bit 4 = 1: reset Data Service Interrupt Pending status bit.

bit 5 = 0: no action
bit 5 = 1: reset Terminate Interrupt Pending status bit.

bit 6 = 0: no action.
bit 6 = 1: reset Ready Change Interrupt Pending status bit.

bit 7 = 0: no action
bit 7 = 1: reset Special Interrupt Pending status bit.

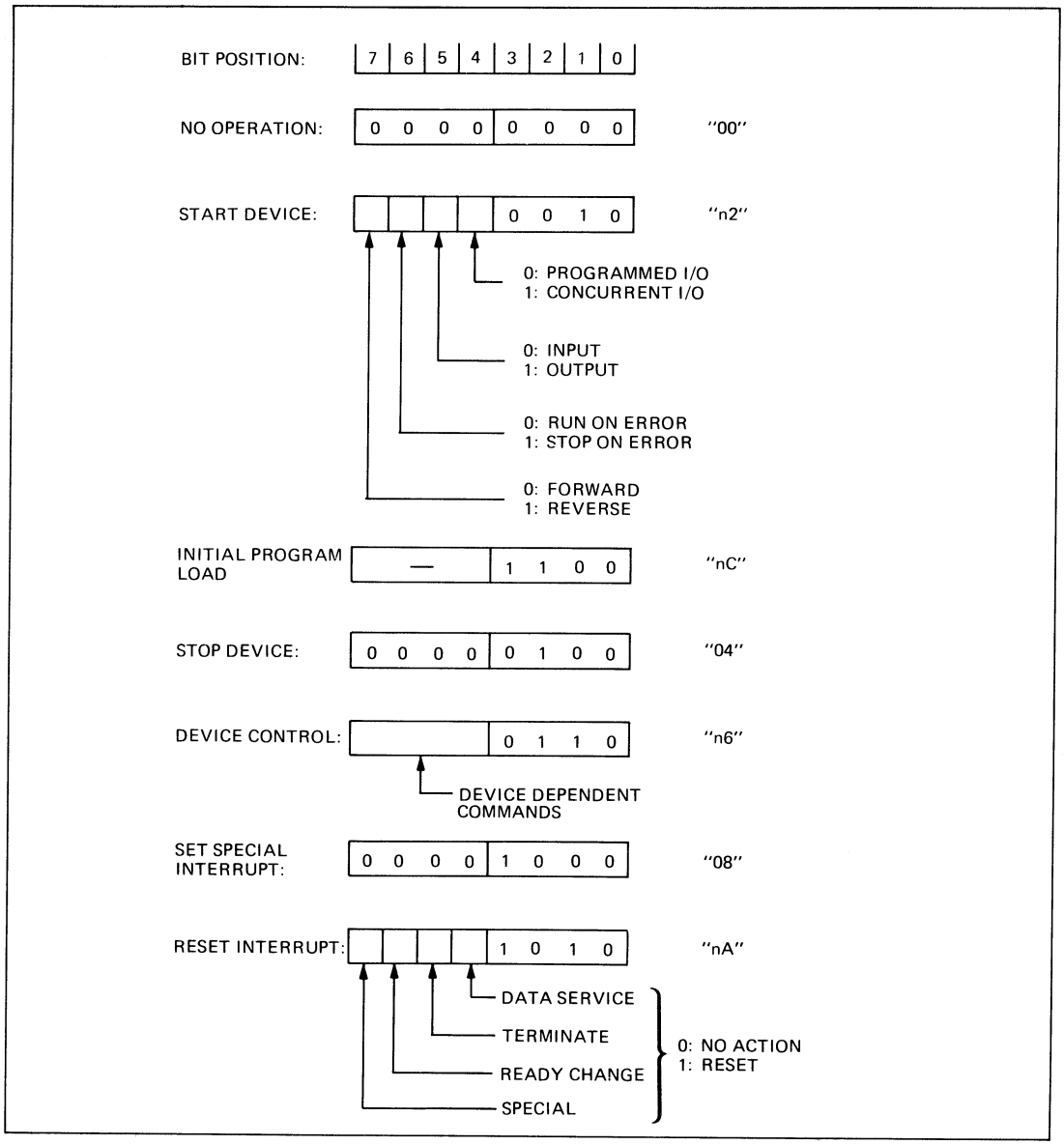


Figure A. Order Code Field of Device Register Block.

4.9 MODE BYTE FORMAT, DEVICE REGISTER BLOCK

The standard format of the DRB's mode byte provides the capability of setting up to four types of operating modes within the controller.

The standard format used in writing the mode byte of the Device Register Block (DRB) is shown in Figure A. It provides the capability to set up to four different types of modes in the controller. The interrupt enable/disable mode is standard to all controllers. The parity check/generate mode is utilized in all controllers which interface to devices which operate with parity bits. The remaining two modes are utilized for device-specific purposes. Note that any one or more modes can be changed without affecting the setting of the other modes.

The format of the byte obtained in reading the DRB mode byte is device dependent. However, bit 8 is always a 0 if in the disable interrupt mode, and a 1 if in the enable interrupt mode.

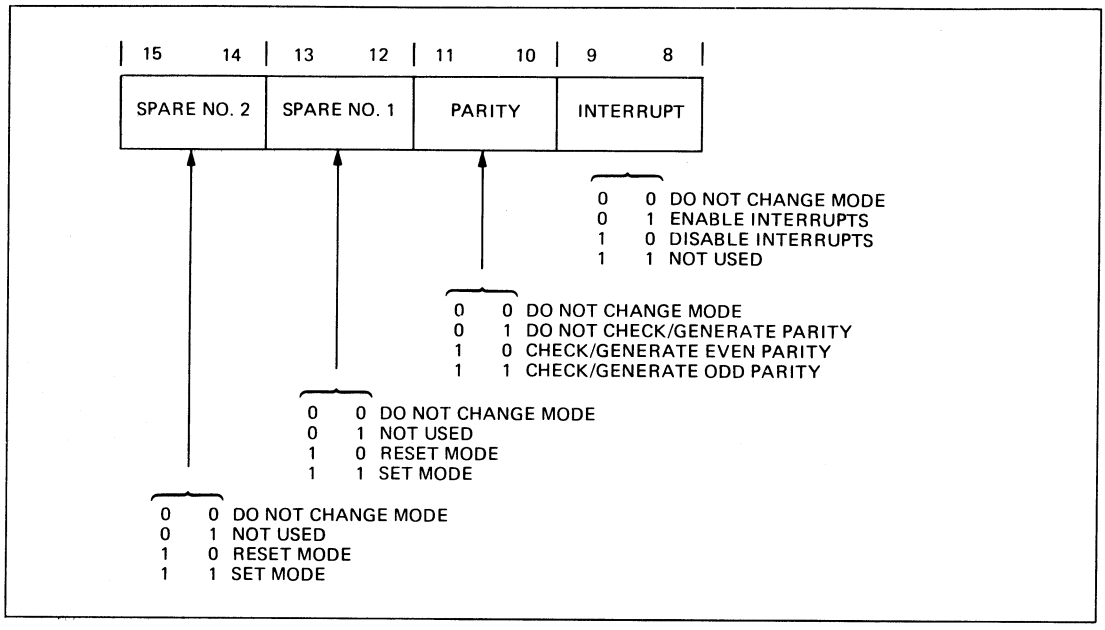


Figure A. Mode Field of Device Register Block.

4.10 INPUT CONTROLLER STATES

A typical controller for an input device operates in four states. This topic defines these states and relates them to DRB status bits.

A typical programmed I/O or concurrent I/O controller for an input device is a four-state machine. The states, transition between states, and the values of the four least significant status bits are shown in Figure A.

The four states are defined as follows:

Rest:	controller is at rest and waiting for a start order;
Wait:	controller has been ordered to move data from the device and is waiting for that data;
Data Ready:	controller is ready to move data to the computer;
Status Update:	controller is in the process of updating its status bits.

The condition $\overline{\text{RUNENABLE}}$ is defined in Figure A. If this condition occurs the controller will exit the wait or data ready states, go into the status update state, and return to the rest state.

Note that the controller will return to the wait state after each data transfer unless the $\overline{\text{RUNENABLE}}$ condition occurs.

The listing of status bits in Figure A assumes that the device remains ready. If the device goes not ready the controller will abort its operation, causing the $\overline{\text{RUNENABLE}}$ condition. It will then return to the rest state. However, the controller cannot exit the rest state until the device goes ready again.

Note that the data service interrupt pending status bit will go true in the data ready state only if the controller is executing a programmed I/O data transfer. This bit will remain true when the c ontroller returns to the wait state. It is only reset by the software issuing a reset interrupt order to the controller.

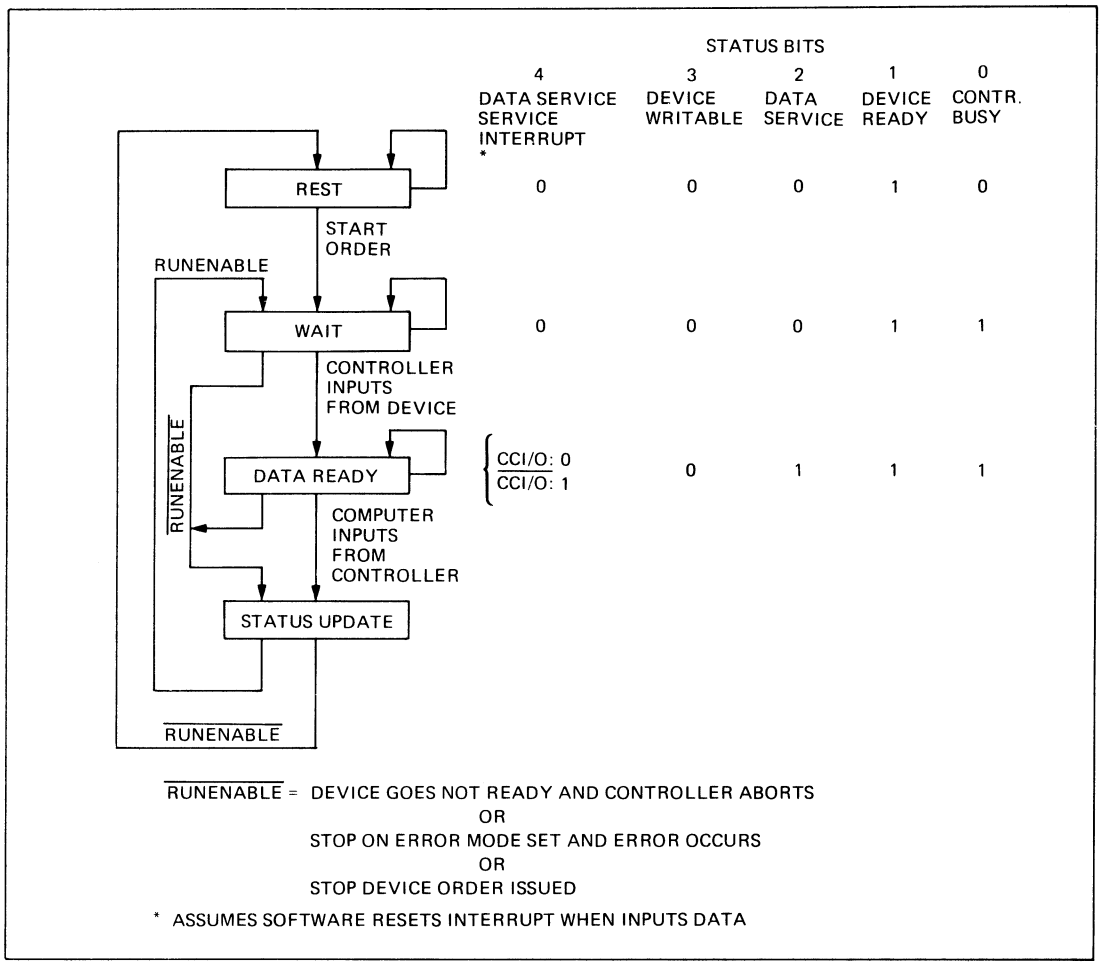


Figure A . State Diagram for Input Controller.

4.11 OUTPUT CONTROLLER STATES

A typical controller for an output device operates in four states. This topic defines the four states and relates them to DRB status bits.

A typical programmed I/O or concurrent I/O controller for an output device is a four-state machine. The states, transition between states, and values of the four least significant status bits are shown in Figure A.

The four states are defined as follows:

Rest:	controller is at rest and waiting for a start order;
Data Ready:	controller has been ordered to move data to a device and is ready for the data from the computer;
Wait:	controller has data from computer and is waiting to move data to device;
Status Update:	controller is in process of updating its status bits.

The condition $\overline{\text{RUNENABLE}}$ is defined in Figure A. If this condition occurs the controller will exit the data ready state, go into the status update state, and return to the rest state.

Note that the controller will return to the data ready state after each data transfer unless the $\overline{\text{RUNENABLE}}$ condition occurs.

The listing of status bits in Figure A assumes that the device remains ready. If the device goes not ready the controller will abort its output operation, causing the $\overline{\text{RUNENABLE}}$ condition. It will then return to the rest state. However, the controller cannot exit the rest state until the device goes ready again.

Note that the data service interrupt pending status bit will go true in the data ready state only if the controller is executing a programmed I/O data transfer. This bit will remain true when the controller leaves the data ready state unless the software issues a reset interrupt order to the controller.

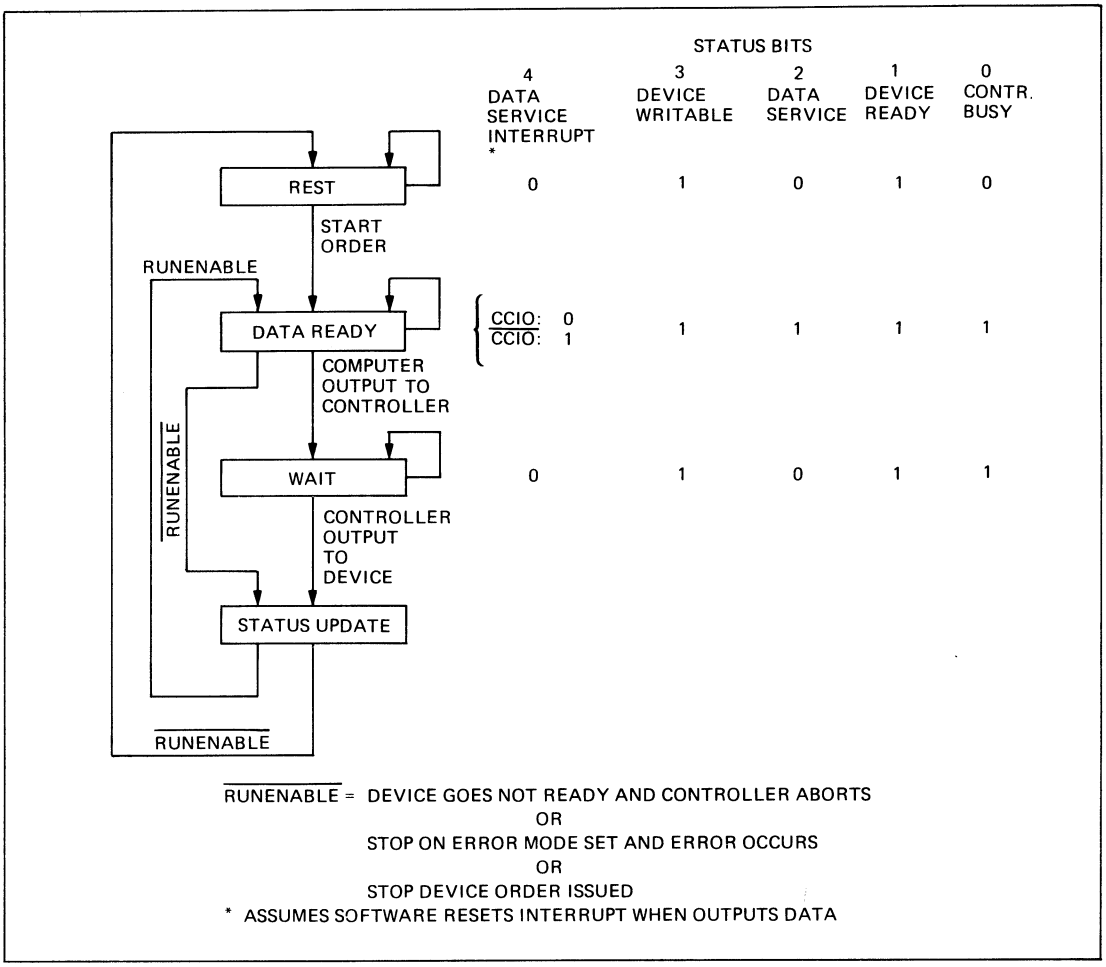


Figure A. State Diagram for Output Controller.

4.12 I/O FOR THE MAINTENANCE FRONT PANEL

With the exception of the hardware status indicators and the leftmost six display selectors, the maintenance front panel switches and displays are read and written as a special purpose I/O device controller. This permits control of the panel by 32/S software.

The switches and displays of the maintenance front panel are read and written, respectively, as a special purpose I/O device controller on the Monobus. The status indicators and the leftmost six display selectors are exceptions; these displays are not program accessible.

The Device Register Block for the maintenance panel has a specialized format, as shown in Figure A. This DRB is located at Monobus word addresses "3FFFC" through "3FFFF". All read and write operations must be word operations.

Reading word location "3FFFC" inputs the following data:

bit 15: 1 if data switch 17 is depressed (leftmost switch).

bit 14: 1 if data switch 16 is depressed (second from left switch)

bits 13-7: 100000_2

bits 7-4: binary encoded indication of which display selector switch is depressed. The switches are weighted in their order on the panel, starting at the left, e.g., the DAR/LOC/M selector is 0001; the DAR/ABS/M selector is 0010, etc.

bit 3: 1 if the keylock is in the LOCK position.

bit 2: 1 if the address enter switch is depressed.

bit 1: 1 if the data enter switch is depressed.

bit 0: 1 if the advance switch is depressed.

Reading word location "3FFFF" inputs the data switches.* A bit is a 1 if the switch is depressed. Bit positions match the switch positions.

Writing word location "3FFFD" sets the leftmost displays of the address display.*

bit 15: if a 1, turns on address display 17.

bit 14: if a 1, turns on address display 16.

Writing word location "3FFFE" sets the least significant 16 bits of the address display. If a bit is a 1, the display is turned on. Bit positions match the display positions.

Writing word location "3FFFF" sets the least significant 16 bits of the data display.* If a bit is a 1, the display is turned on. Bit positions match the display positions. The two most significant bits of the data display cannot be turned on.

Note that the conventional definition of word address versus byte address does not apply to the maintenance panel's DRB.

*These operations can only be performed by the processor. The 32/S software cannot read or write words at odd addresses.

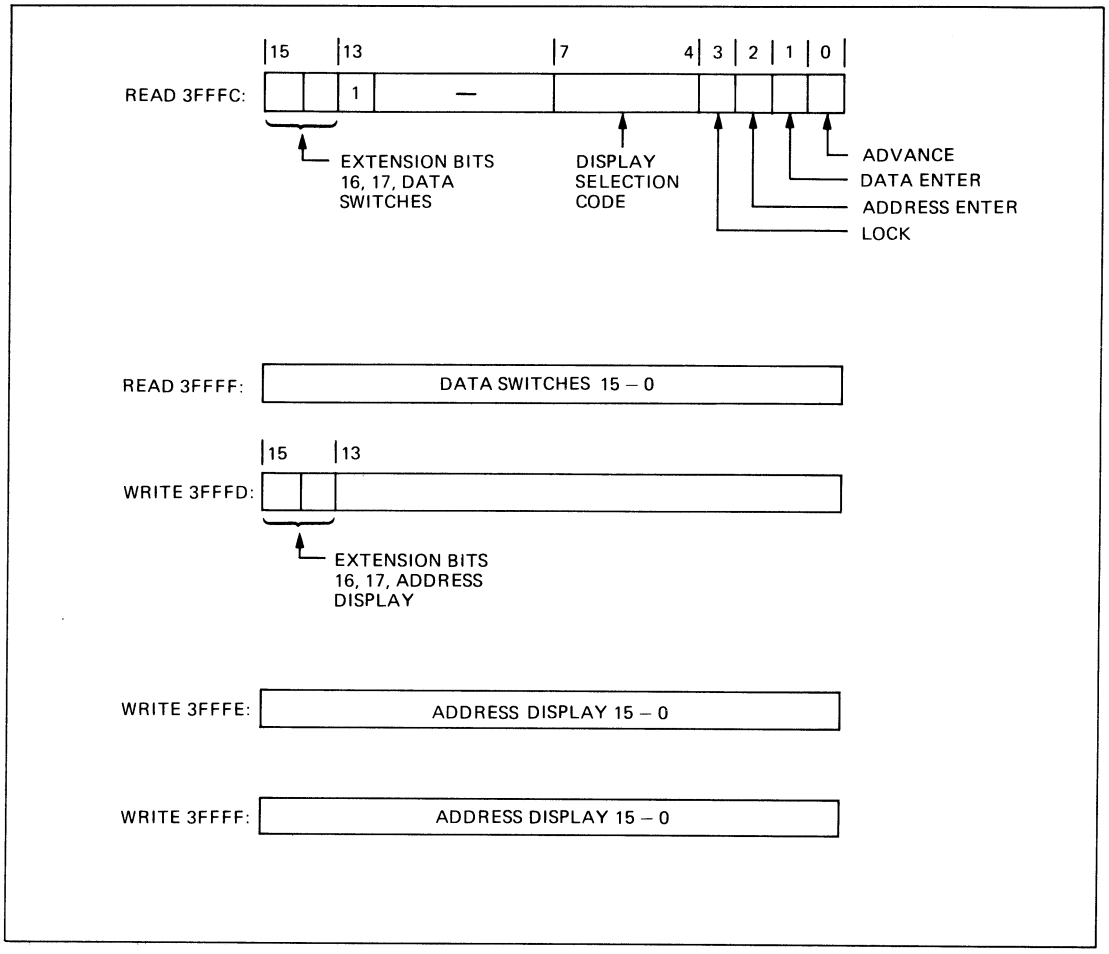


Figure A. Special Device Register Block for Maintenance Front Panel.

5 Data

5.1 DATA FORMAT LENGTHS

Data format lengths of variable field, byte, word, doubleword, and tripleword are defined at Monobus Locations. Operations within the top of the stack are defined for word, doubleword, and tripleword length data.

Data of variable field length, byte, word, doubleword, or tripleword size can be loaded into the top of the stack from Monobus locations. Similarly, variable field length, byte, word, doubleword, or tripleword size data can be stored into Monobus locations from the top of the stack. Field and byte data are always right justified as they are loaded into TOS, and the remaining most significant bit positions are set to zeroes. Field and byte data are extracted from the rightmost field or byte of TOS when they are to be stored into a Monobus location. See Figure A.

The standard 32/S instruction set provides for operations on word, doubleword, and tripleword size data within the top of the stack.

Variable length field data is defined by a field descriptor word. The least significant four-bit field specifies the bit position of the rightmost bit of the field within the Monobus word. The next most significant four-bit field specifies the field length, minus one. (See topic 7.11). When a field is stored into a Monobus word location the bit positions outside of the field are left undisturbed. When a field is loaded into TOS, the bit positions to the left of the field are set to zeroes. See Figure B.

The location of byte data within a Monobus word location is specified by the least significant bit of the Monobus address. A least significant bit value of zero specifies the most significant, or leftmost byte within the word. When a byte is stored into a Monobus word location the other byte of the word is left undisturbed. When a byte is loaded into TOS, the leftmost eight bits are set to zeroes.

The Monobus address of a doubleword points to its most significant 16 bits, DW1. The least significant 16 bits, DW0, are in the next higher word address location. When loaded into the top of the stack, DW1 is pushed first, and DW0 is pushed next, leaving DW0 in TOS and DW1 in TOS1.

The Monobus address of a tripleword points to its most significant 16 bits, TW2. The next most significant 16 bits, TW1, are in the next higher word location, and the least significant 16 bits, TW0, are in the next higher word location. When loaded into the top of the stack, TW2 is pushed first, TW1 next, and TW0 last, leaving TW0 in TOS, TW1 in TOS1, and TW2 in TOS2.

Numeric data formats are defined for word, doubleword, and tripleword length data. Logic value data formats are defined as word and doubleword length data. These formats are defined in topic 5.2

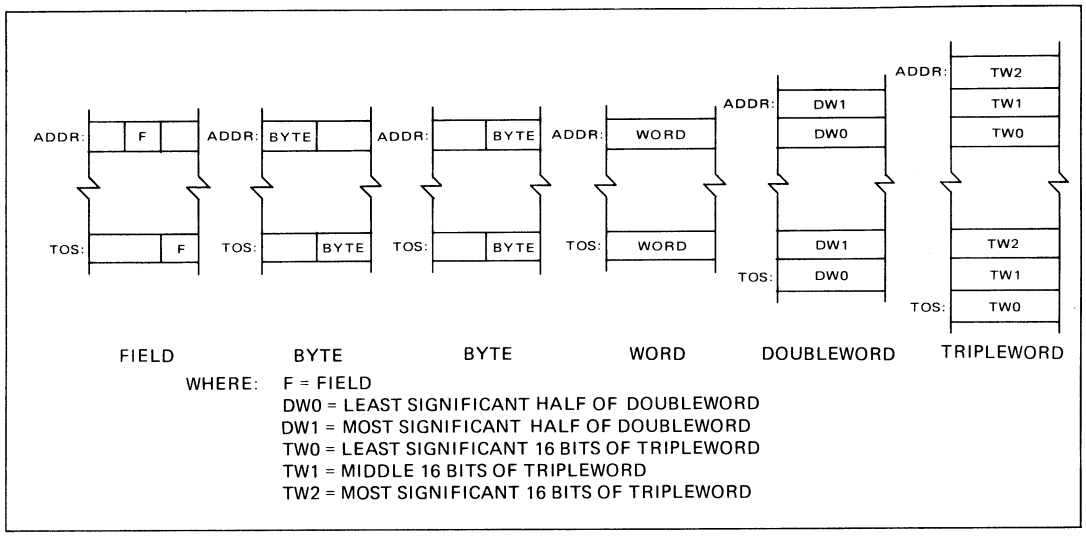


Figure A. Data Formats, Monobus Location and TOS.

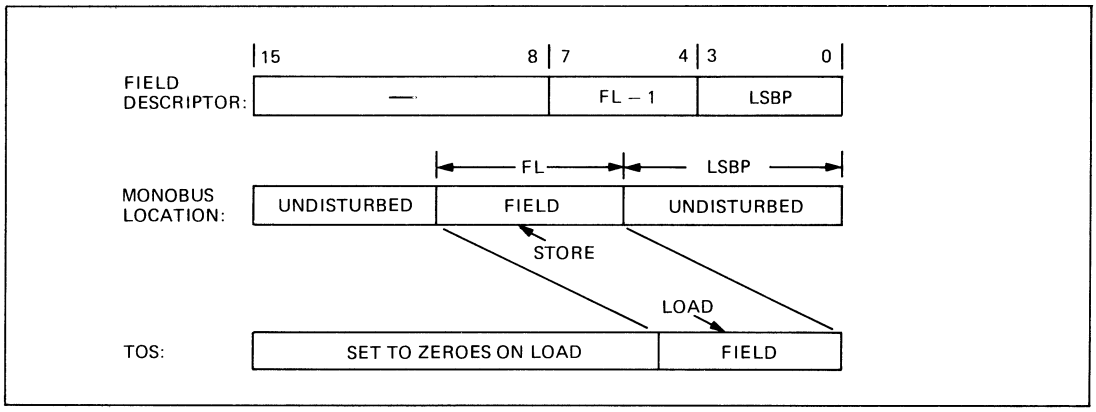


Figure B. Relationship Between Field Formats, Monobus Location and TOS.

5.2 NUMERIC AND LOGICAL DATA FORMATS

Formats are defined for integer and floating point numeric data, and for logical data.

The 32/S instruction set can process binary integers of word and doubleword length, floating point numbers of tripleword length, and logical information of word and doubleword length. The operations performed fall into three classes: signed integer arithmetic, floating point arithmetic, and logical operations.

Integer Arithmetic Operations

The operands for integer arithmetic operations may be either word or doubleword signed numbers. See Figure A. Negative numbers are represented in two's complement form. For word operations, the two word operands are contained in TOS and TOS1 before the operation begins. For doubleword operations, the two operands are contained in TOS, TOS1 and TOS2, TOS3 with the most significant portion of the operands contained in TOS1 and TOS3. Arithmetic operations are performed by popping the two operands from the stack and combining them according to the operation being performed. The result is then pushed into the stack.

The operation of addition can also be performed on word operands using TOS as one operand and a word in memory as the other operand. This operation pops the TOS operand and adds it to the memory operand. The sum replaces the memory operand.

Floating Point Arithmetic Operations

Floating point operands are tripleword in length. See Figure B.

The fraction of a floating point number is expressed in base 1000 digits, each digit consisting of 10 binary bits having the value 0-999. The fraction consists of four digits, thus giving a precision for normalized numbers of from 10 to 13 decimal places. The fraction is represented in true magnitude with the fraction sign in bit position 47. Specifically:

S = 0: positive fraction

S = 1: negative fraction

The radix point of the fraction is assumed to be to the left of the most significant fraction digit. To represent the magnitude of the floating point number, the fraction is considered to be multiplied by a power of 1000. The characteristic, CHAR, occupies bit positions 40-46 of the floating point number, and indicates the power of 1000. The characteristic is treated as an excess 64 number with a range -64 through +63 and thus permits the representation of numbers in the range $.1 \times 10^{-192}$ to $.999999999999 \times 10^{191}$.

All floating point numbers are considered to be in normalized form, that is, the most significant fraction digit is not zero. The only exception is the floating point zero. A floating zero is represented by each of the three words of the number being zero.

If any digit of the fraction is greater than 999, the floating point number is considered undefined. The result of any floating point operation where one or both operands is undefined is also undefined. If any operation causes exponent overflow, the first word of the result is set to all binary one's (this is an undefined result). Division by a floating zero will also result in the first word of the result set to all binary one's.

The operands for floating operations are contained in TOS, TOS1, TOS2 and TOS3, TOS4, TOS5. The operations are performed by popping the two operands from the stack and combining them according to the operation being performed. The result is then pushed into the stack. The result of a floating point operation may be a floating point number, word integer, or a doubleword integer.

Logical Operations

The operands for logical operations are word or doubleword quantities. The two word operands are contained in TOS and TOS1. The two doubleword operands are contained in TOS, TOS1 and TOS2, TOS3 before the operation begins. The operations are performed by popping the two operands from the stack and combining them according to the operation being performed. The result is then pushed into the stack. For logical operations each corresponding pair of operand bits is combined independently of the other bits.

The result of executing a comparison instruction (see topic 7.5) is to leave a True/False word result in the top of the stack. See Figure C. The least significant bit defines the True or False value, and the most significant 15 bits are zeroes. Specifically:

bit 0 = 1: True

bit 0 = 0: False

bits 15-1: All zeroes.

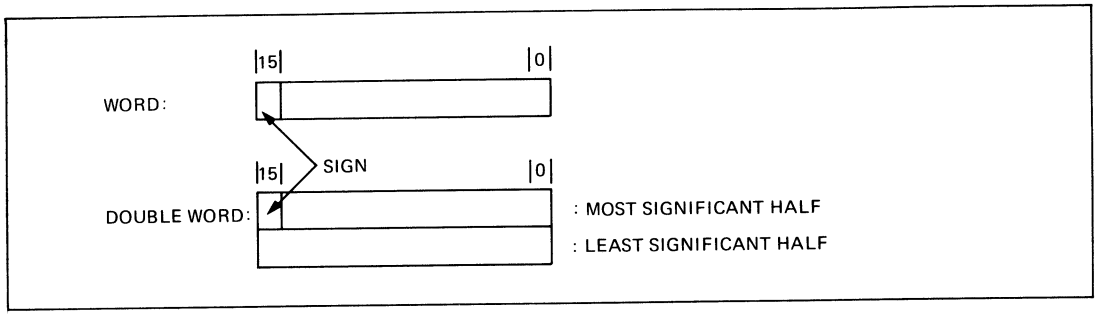


Figure A. Numeric Word and Doubleword Formats.

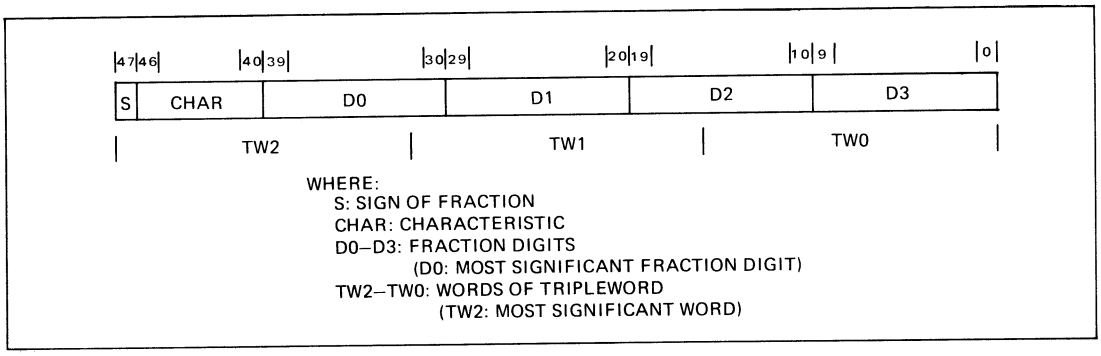


Figure B. Floating Point Numeric Format

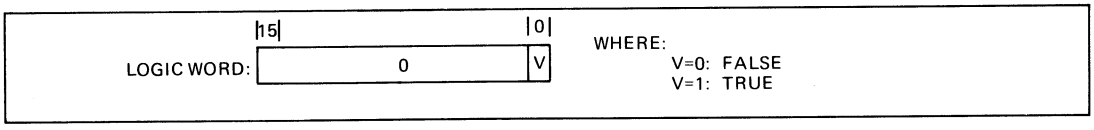


Figure C. Logic Value Word Format.

6 Memory Reference Instructions

6.1 MEMORY REFERENCE INSTRUCTIONS - INTRODUCTION

The 15 types of memory reference instructions perform load and store operations on each of the different size data items, add and subtract word to the data stack operations, and the add word to memory operation. Three formats are used to accommodate eight addressing modes for each of the 15 types of instructions.

The memory reference instructions transfer data between an addressed location on the Monobus and the top of the stack. The addressed location is specified by an addressing mode in the instruction and calculated from a displacement field in the instruction and index, indirect, and base addresses in the top levels of the stack. There are three formats for these instructions, differing in the size of displacement field required for the addressing mode.

The instruction formats are shown in Figure A. Each of the three formats is identical in the first byte, having a 1 in the most significant bit position, followed by a four-bit operation code field, followed by a three-bit addressing mode field. Format 1 contains no address displacement field; Format 2 contains an 8-bit address displacement field, D8; Format 3 contains a 16-bit address displacement field, D16.

The 15 types of memory reference instructions are listed in Figure B. Store and load instructions transfer word operands (integer numbers and logic values), doubleword operands (integer numbers and logic values), and tripleword operands (floating point numbers). Loading a byte operand pushes the byte into the stack as a word operand, with the least significant 8-bits of the word containing the byte and the most significant 8-bits set to zeroes. Storing a byte operand stores the least significant 8-bits of the word in the top of the stack into a byte of memory. Loading a field operand pushes a specified field of 1 to 16 bits into the stack as a word operand. The field is right-justified in the word and the remaining (leftmost) bits are set to zeroes. Storing a field operand stores a specified field of 1 to 16 bits of the word in the top of the stack into a specified portion of a word in memory. All store instructions except the store word nondestructive pop the word(s) containing the operand from the stack.

Add and subtract instructions add and subtract the word from the Monobus location to/from the word in the top of the stack. A second type of add instruction adds the word in the top of the stack in with the word in the Monobus location. The swap type of instruction interchanges the word in the top of the stack with a word in memory.

The general categories of memory addressing are diagrammed in Figure C. The absolute mode provides a full 18-bit addressing range displacement from a base value contained in TOS1. This mode is used to address Monobus locations outside of a data stack, such as I/O controller Device Register Blocks, alterable control memory locations, and system tables (e.g., Program Library, Concurrent I/O Control Block). It may only be used while operating in executive mode.

The global addressing modes provide a 16-bit addressing range displacement from the Stack Base, SB. These modes are used to address variables which are global to an MPL PROCEDURE.

The local addressing modes provide an 8-bit addressing range displacement from the Environmental Pointer, EP. These modes are used to address variables which are local to the current environment.

The constant addressing mode provides a 16-bit addressing range displacement from the Program Base, PB. This mode is used to address constants stored within the program segment.

The addressing modes are specified in topic 6.2. The instruction types are defined in topics 6.3, 6.4, 6.5, and 6.6.

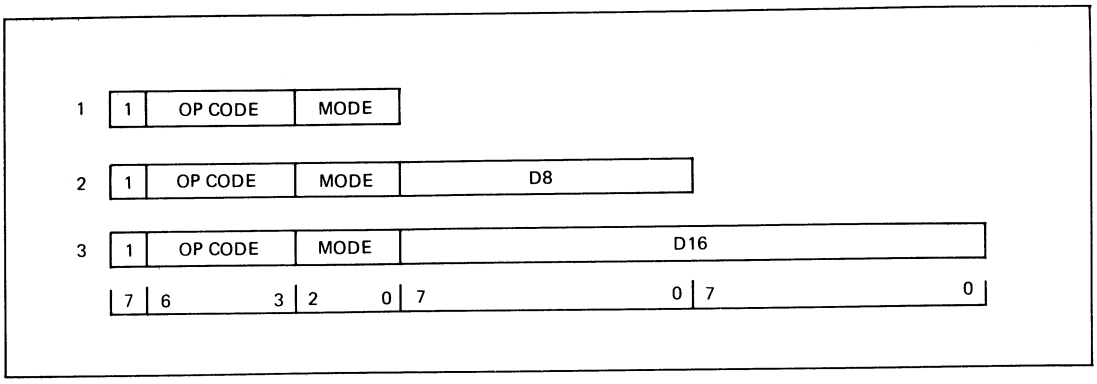


Figure A. Memory Reference Instruction Formats.

STORE	FIELD
	BYTE
	WORD
	WORD, NONDESTRUCTIVE
	DOUBLE WORD
	TRIPLE WORD
LOAD	FIELD
	BYTE
	WORD
	DOUBLE WORD
	TRIPLE WORD
ADD	WORD TO MEMORY
	WORD (TO STACK)
SUBTRACT	WORD (FROM STACK)
SWAP	WORD (WITH MEMORY)

Figure B. Op Codes.

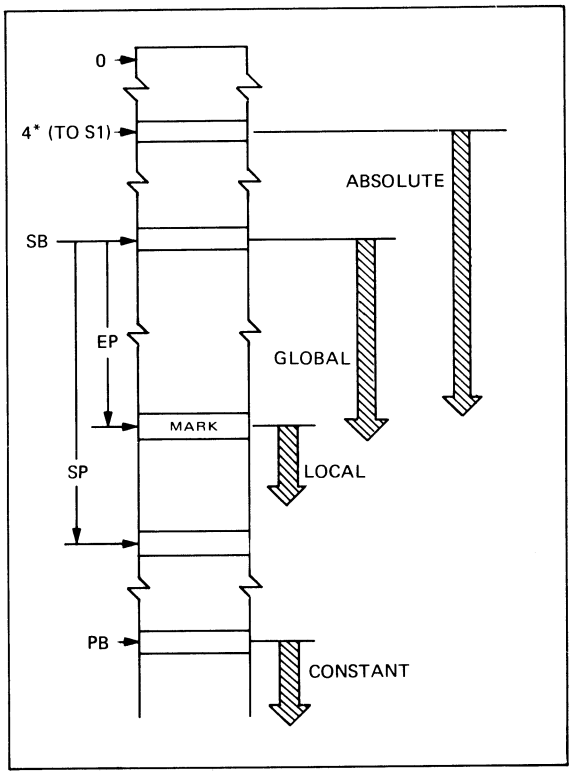


Figure C. Addressing Modes.

6.2 ADDRESSING MODES

Eight addressing modes are defined for the memory reference instructions. This topic defines the effective address calculation for each mode.

The addressing mode specifies the calculation to be performed in determining the effective address for accessing the Monobus. The effective address expression for each addressing mode is listed in Figure A, along with the use of the mode and the instruction format required. The effective address expressions are also shown graphically in Figure B.

Definitions of the terms used in the effective address expressions are:

SB:	Stack Base register value, 18 bit absolute address of base of data stack.
EP:	Environmental pointer register value, 16 bit address of the base of the current Mark, relative to SB.
D8:	8-bit address displacement field of instruction.
D16:	16-bit address displacement field of instruction.
TOS(X):	16-bit index contained in TOS, the top level of the stack. This index specifies a number of data items, independent of data length; e.g., number of bytes, number of words, etc. It is converted to a byte-level index when the memory reference instruction is executed. For example, if an indexed doubleword instruction is executed, the index value is multiplied by 4.
TOS(D16):	16-bit address displacement contained in TOS, the top level of the stack.
TOS1(D16):	16-bit address displacement contained in TOS1, the second to top level of the stack.
TOS1(D18):	18-bit base address contained in TOS1, the second to top level of the stack. NOTE: Only the most significant 16 bits of the D18 displacement are stored in TOS1; the least significant 2 bits are assumed to be zeroes. The TOS1(D18) is multiplied by 4 (as shown for mode 7) when calculating the effective address.

As shown graphically in Figure B, the effective addresses of modes 0, 1, 4 and 5 are based at SB, the Stack Base. Mode 0 provides direct addressing and mode 1 provides direct addressing with indexing via the contents of TOS. Mode 4 provides indirect addressing via the contents of TOS. Mode 5 provides indirect addressing via the contents of TOS1, with indexing via the contents of TOS.

The effective addresses of modes 2 and 3 are both based at SB + EP, the absolute address of the base of the latest Mark in the stack. Mode 2 provides direct addressing and Mode 3 provides direct addressing with indexing via the contents of TOS.

The effective address of Mode 6 is based at the Program Base, PB. It provides direct addressing with indexing via the contents of TOS. Mode 6 may not be used with store instructions, AWM, or SWAP. An interrupt (see topic 3.3) will be generated if so used.

The effective address of Mode 7 is based at address contained in TOS1 (multiplied by 4, as TOS1 contains the most significant 16 bits of the 18-bit base address). It provides indirect addressing via the contents of TOS1, with indexing via the contents of TOS. It is a privileged mode.

All effective address calculations are performed only on the least significant 16-bits of the 18-bit addresses. The upper 2 bits of the 18-bit base address are not modified during the address calculations; therefore, all effective addresses "wraparound" at the 64K byte bank boundaries (see topic 2.1).

For the store and add to memory instructions, the references to TOS and TOS1 in the above effective address expressions refer to the stack after the data item to be stored, or added, to memory has been popped from the data stack.

ADDRESSING MODE	EFFECTIVE ADDRESS	USE	INSTRUCTION FORMAT
0	SB + D16	GLOBAL DIRECT	3
1	SB + D16 + TOS(X)	GLOBAL DIRECT, INDEXED	3
2	SB + EP + D8	LOCAL DIRECT	2
3	SB + EP + D8 + TOS(X)	LOCAL DIRECT, INDEXED	2
4	SB + TOS(D16)	INDIRECT THRU TOS	1
5	SB + TOS(X) + TOS1 (D16)	INDIRECT THRU TOS, INDEXED	1
6	PB + D16 + TOS(X)	CONSTANT DIRECT, INDEXED	3
7	TOS(X) + 4*TOS1 (D18)	ABSOLUTE, INDEXED	1

Figure A. Addressing Modes and Effective Addresses.

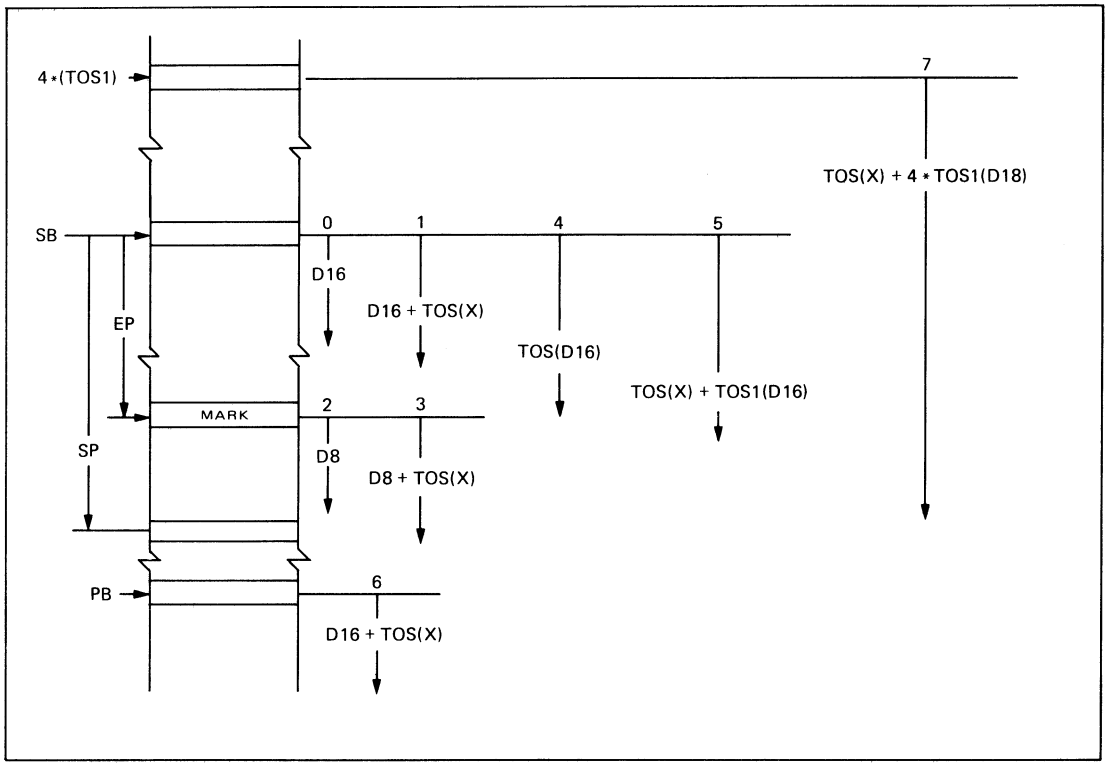


Figure B. Addressing Diagram.

6.3 STORE INSTRUCTIONS

Store instructions are defined to store field, byte, word, doubleword, and tripleword data.

There are six types of store instructions. Each of these instructions may have any of the three formats shown in topic 6.1, with any of the corresponding address modes shown in topic 6.2.

Figure A shows the data stack before and after execution of five of the six types of store instructions (the STT is similar to STD). There are five "before" pictures of the stack for each instruction, each labeled (across the top) with the address modes for which they apply. As can be seen from the effective address expressions (see topic 6.2), the stack must, for some modes, contain base address, index, and indirect address values in the stack below the item to be stored. The references to TOS and TOS1 in the effective address expressions refer to the stack after the data item to be stored has been popped from the stack.

Note that, not only the data item to be stored, but also the effective address parameters (X, D16, D18) are popped from the stack in the execution of the store instructions. The one exception is the Store Word Non-Destructive in which the stored word is pushed back into the stack at the conclusion of execution.

STF Store Field — Op Code "1"

A field descriptor word, FD, contained in TOS1 is used to insert the field, F, contained in TOS into the word at the effective address. The bits outside the field are not changed. The field descriptor specifies the length of the right-justified field F in TOS and the bit position to insert F in the word in memory. The definition and interpretation of the field descriptor are given in topic 7.11.

The field descriptor, FD, and the word containing the field, F, are popped. The effective address parameters are popped.

STB Store Byte — Op Code "7"

The 8-bit byte, B, contained in the least significant 8 bits of TOS, is stored at the effective address. The word containing the byte is popped. The effective address parameters are popped.

STW Store Word — Op Code "6"

The word, W, contained in TOS is popped and stored in the effective address. The effective address parameters are popped.

STWN Store Word Non-Destructive — Op Code "5"

The word, W, contained in TOS is popped and stored in the effective address. The effective address parameters are popped. A copy of the word W is then pushed back into the stack.

STD Store Doubleword — Op Code "0"

The doubleword on the top of the stack is popped and stored at the effective address. Specifically, TOS1 is stored in the word at the effective address and TOS is stored at the effective address, plus two.

The effective address parameters are popped from the stack.

STT Store Tripleword — Op Code "3"

The tripleword on the top of the stack is popped and stored at the effective address. Specifically, TOS2 is stored into the word at the effective address. TOS1 is stored into the word at the effective address plus 2, and TOS is stored into the word at the effective address plus 4.

The effective address parameters are popped from the stack.

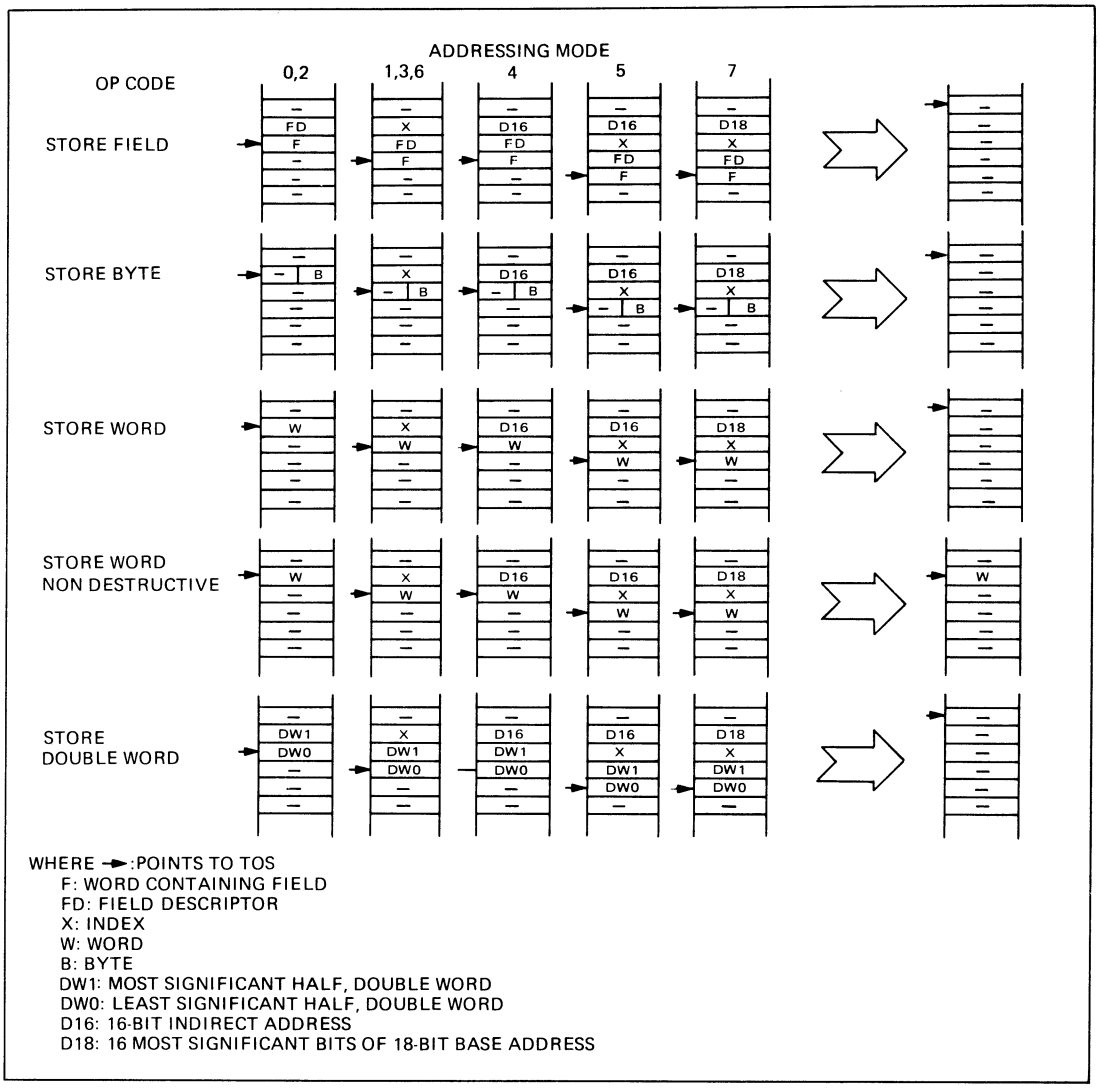


Figure A. Memory Before and After Execution of Store Instructions.

6.4 LOAD INSTRUCTIONS

Load instructions are defined to load field, byte, word, doubleword, and tripleword data.

There are five types of load instructions. Each of these instructions may have any of the three formats shown in topic 6.1, with any of the corresponding addressing modes shown in topic 6.2.

Figure A shows the data stack before and after execution of four of the five types of load instructions. (The LTW is similar to LD). There are five "before" pictures of the stack for each instruction, each labeled (across the top) with the address modes for which they apply. As can be seen from the effective address expressions (topic 6.2), the stack must, for some modes, contain base address, index and indirect address values before the load instructions are performed. The effective address parameters are popped before the data item is pushed into the stack.

LF Load Field — Op Code: "A"

A field descriptor, FD, is popped and used to extract a field, F, from the word at the specified address. The effective address parameters are popped. The extracted field is right-justified and pushed into the stack; if the field is less than 16 bits long the bits in the stack to the left of the field are set to zeroes. The definition and interpretation of the field descriptor are given in topic 7.11.

LB Load Byte — Op Code: "F"

The effective address parameters are popped. An 8-bit byte, B, obtained from the effective address, is pushed into the stack. The byte is right-justified and the most significant 8 bits are set to zeroes.

LW Load Word — Op Code: "E"

The effective address parameters are popped. The word at the effective address is pushed into the stack.

LD Load Doubleword — Op Code "8"

The effective address parameters are popped. The doubleword at the effective address is pushed into the stack. Specifically, the word at the effective address is loaded into TOS1, and the word at the effective address plus 2 into TOS.

LTW Load Tripleword — Op Code "D"

The effective address parameters are popped. The tripleword at the effective address is pushed into the stack. Specifically, the word at the effective address is loaded into TOS2, the word at the effective address + 2 into TOS1, and the word at the effective address + 4 into TOS.

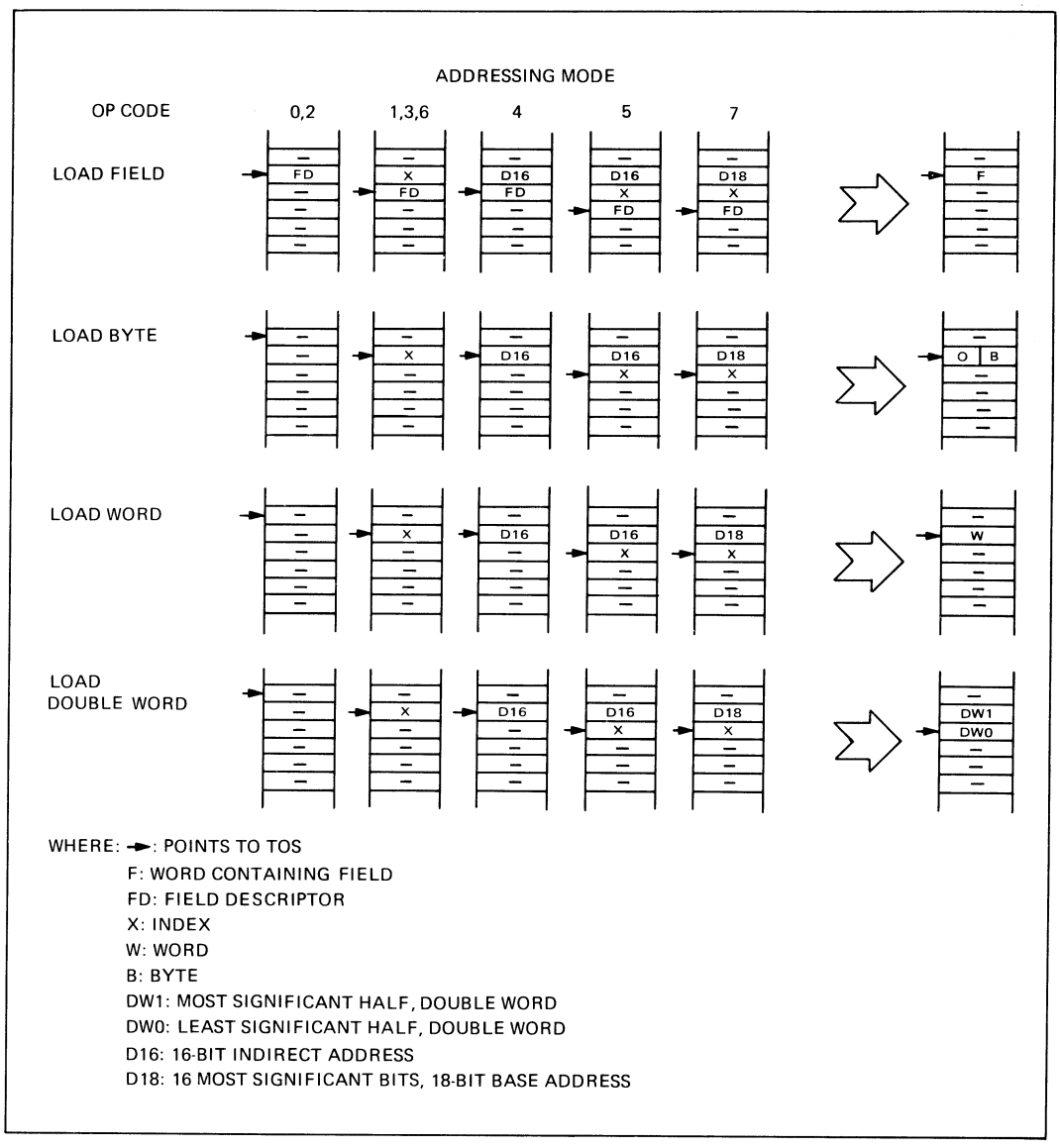


Figure A. Memory Before and After Execution of Load Instructions.

6.5 MEMORY REFERENCE ARITHMETIC INSTRUCTIONS

Arithmetic instructions are defined to add a word to memory, to add a word to TOS, and to subtract a word from TOS.

There are three types of memory reference arithmetic instructions. Each of these may have any of the three formats shown in topic 6.1, with any of the corresponding addressing modes shown in topic 6.2.

Figure A shows the data stack before and after execution of each of the three types of memory reference arithmetic instructions. There are five "before" pictures of the stack for each instruction, each labeled (across the top) with the address modes for which they apply. As can be seen from the effective address expressions (topic 6.2), the stack must, for some modes, contain base address, index, and relative address values before the arithmetic instructions are performed. These address parameters are popped after adding TOS to memory and before adding or subtracting the memory word to/from TOS.

In each of these three instruction types, a carry result sets or resets the carry bit of the Program Status Register; an overflow result sets the overflow bit of the Program Status Register.

AWM Add Word to Memory Op Code: "4"

The word operand in TOS is added to the word operand at the effective address, and the sum is stored into the word at the effective address. The word in TOS and the effective address parameters are popped.

AW Add Word to Stack Op Code: "C"

The word operand in TOS is added to the word operand at the effective address. The word in TOS and the effective address parameters are popped, and the sum is pushed into the stack.

SW Subtract Word from Stack Op Code: "D"

The word operand at the effective address is subtracted from the word operand in TOS. The word in TOS and the effective address parameters are popped, and the difference is pushed into the stack.

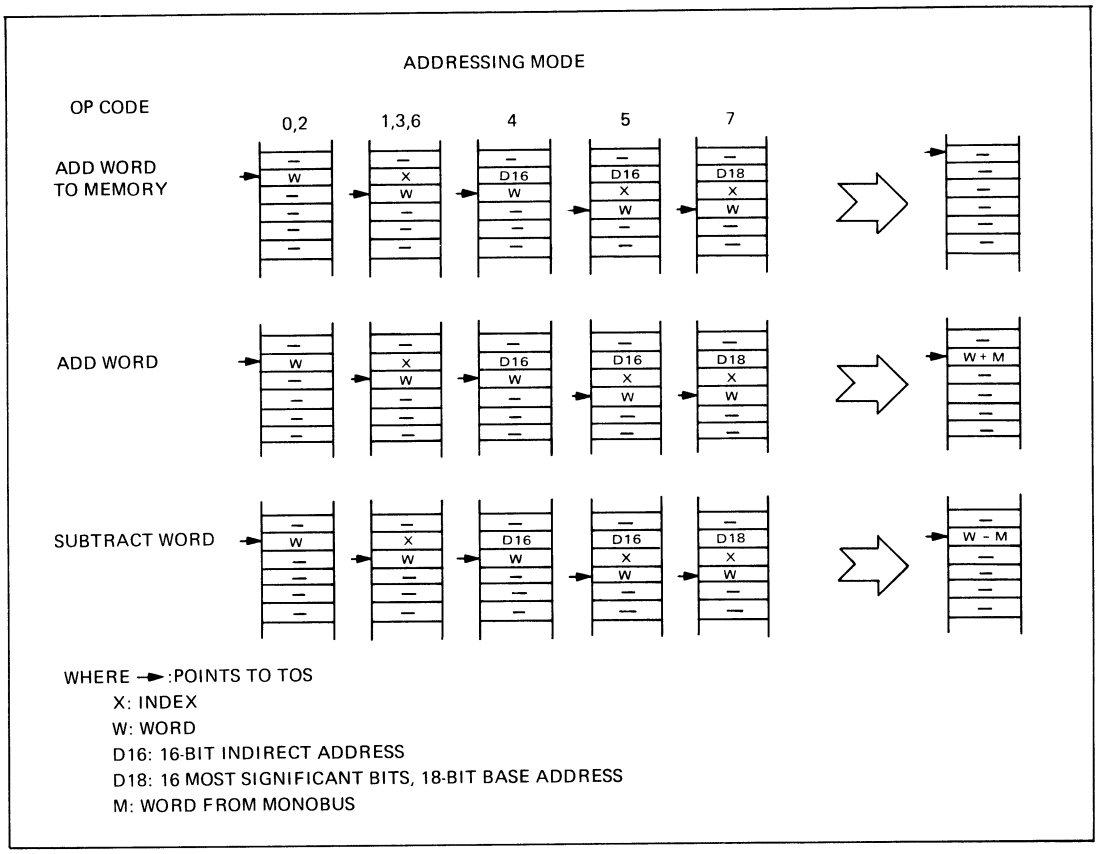


Figure A. Memory Before and After Execution of Memory Reference Arithmetic Instructions.

6.6 MEMORY REFERENCE SWAP INSTRUCTION

An instruction is defined to swap a word in TOS with a word in memory in one uninterruptable memory cycle.

The swap type of instruction swaps the word in TOS with a word in memory in a single memory cycle. The interchange of the two words, since it occurs in a single memory cycle, is guaranteed to be executed without interruption. This instruction provides the capability of synchronizing two processors on the Monobus.

SWAP Swap Word in Stack with Memory Op Code "2"

The word in TOS and the effective address parameters are popped. The word at the effective address is pushed into the stack, and the word which had been in TOS is stored at the effective address.

7 Stack Operate Instructions

7.1 STACK OPERATE INSTRUCTIONS - INTRODUCTION

The stack operate instructions operate on one or two operands in the top of the stack, or push a literal operand into the stack.

The stack operate instructions may be categorized as follows:

arithmetic, word operand

arithmetic, doubleword operand

logical

comparison, arithmetic word and doubleword, logical, floating

shift word and doubleword

load literal and enter configuration switches.

stack modification

field descriptor generation

The stack operate instruction formats are shown in Figure A. The top two formats, which consist only of a single byte or two-byte operation code, are used for all instructions except the load literal instructions. All operands for the instructions using these two formats are in the top of the stack before the instruction is executed. The result produced by the instruction execution is left in the top of the stack.

The execution of the single-byte and two-byte, operation-code-only instructions is illustrated in Figures B and C. Figure B shows the stack before and after execution of a subtract instruction and before and after execution of a less-than comparison instruction. Figure C shows the stack before and after execution of a shift instruction.

The lower five formats in Figure A are used for the load literal instructions. The L field is the literal to be pushed into the stack. There is one instruction type for each format; the difference between types being the length of the literal field. The last format contains a word count field which specifies the word length of the following literal field. The last format may be used to load a floating point literal into the stack.

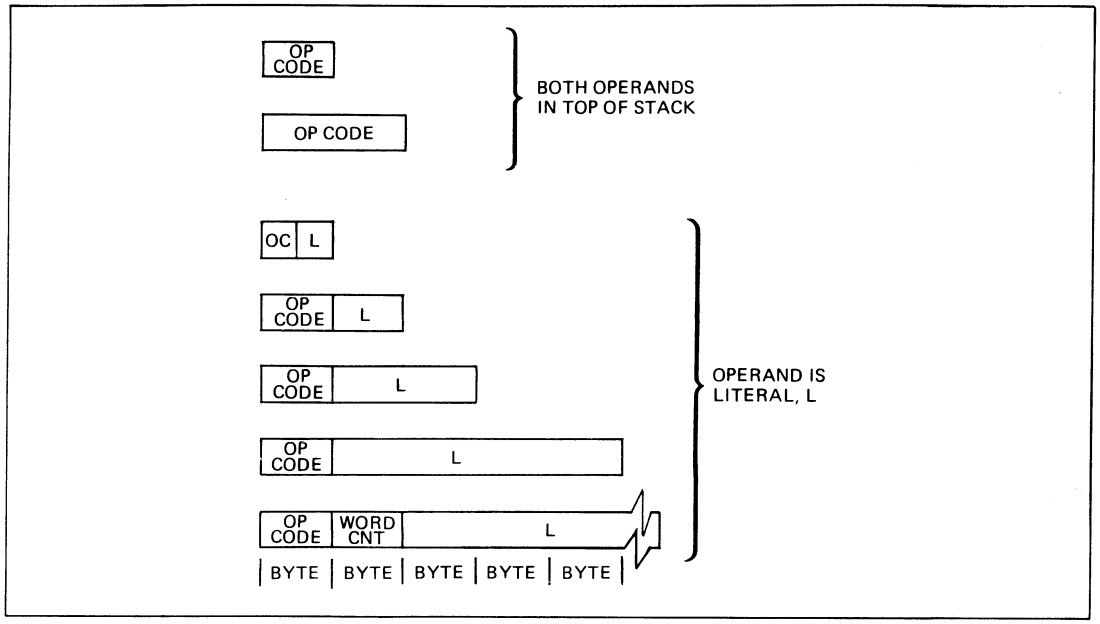


Figure A. Stack Operate Instruction Formats.

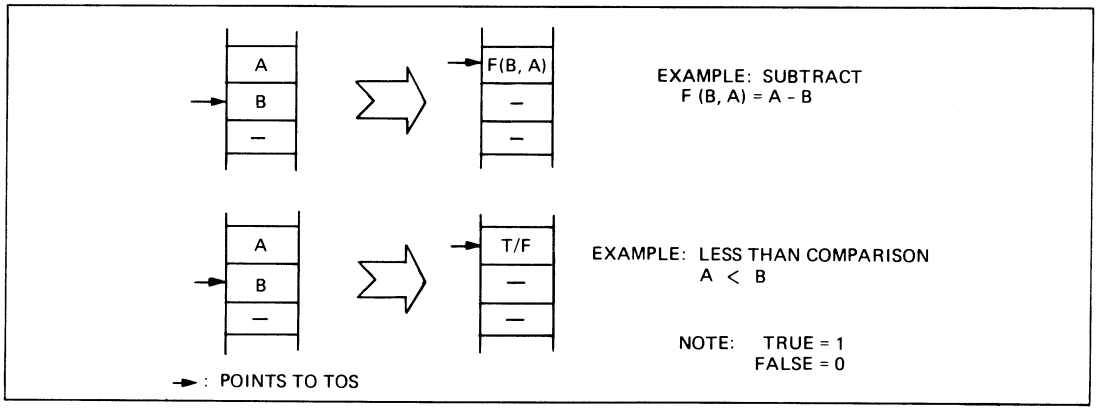


Figure B. Stack Before and After Execution of Arithmetic and Comparison Instructions.

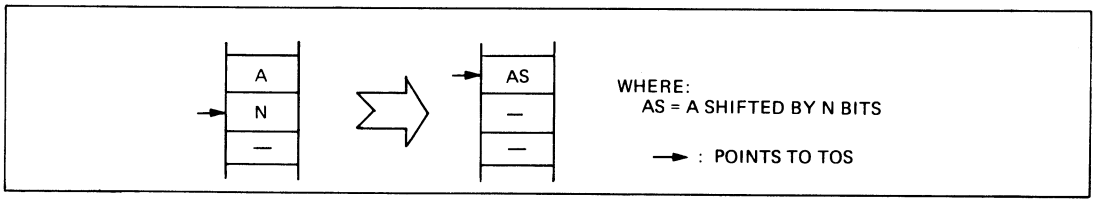


Figure C. Stack Before and After Execution of Shift Instruction.

7.2 ARITHMETIC INSTRUCTIONS, WORD OPERAND

Eight stack operate arithmetic instructions are defined for word operands.

ADD Add — Op Code: "20"

The word operand in TOS is added algebraically to the word operand in TOS1. Both operands are popped from the stack, and the sum is pushed into the stack. The carry and overflow bits of the Program Status Register reflect the results of this operation.

SUB Subtract — Op Code: "21"

The word operand in TOS is subtracted from the word operand in TOS1. Both operands are popped from the stack and the difference is pushed into the stack. The carry and overflow bits of the Program Status Register reflect the results of this operation.

NEG Negate — Op Code: "10"

The word operand in TOS is replaced by its two's complement.

ABS Absolute Value — Op Code: "1E"

If the word operand in TOS is negative, it is replaced by its two's complement. If it is positive, it is left unchanged.

MUL Integer Multiply — Op Code: "22"

The word operand in TOS1 is multiplied by the word operand in TOS. Both operands are popped from the stack. The least significant 16 bits of their product is pushed into the stack.

MULD Multiply with Doubleword Product — Op Code: "36"

The word operand in TOS1 is multiplied by the word operand in TOS. Both operands are popped from the stack. The doubleword product is pushed into the stack. The least significant half of the doubleword is left in TOS and the most significant half in TOS1.

DIV Integer Word Divide — Op Code: "23"

The word operand in TOS is divided into the word operand in TOS1. Both operands are popped from the stack, and the 16-bit integer quotient is pushed into the stack.

MOD Modulo — Op Code: "24"

The word operand in TOS is divided into the word operand in TOS1. Both operands are popped from the stack, and the 16-bit integer remainder is pushed into the stack.

7.3 ARITHMETIC INSTRUCTIONS, DOUBLEWORD OPERAND

Nine stack operate arithmetic operations are defined for doubleword operands.

DADD Doubleword Add — Op Code: "4F00"

The doubleword operand in TOS and TOS1 is added algebraically to the doubleword operand in TOS2 and TOS3. Both doubleword operands are popped and the doubleword sum is pushed into the stack. The carry and overflow bits of the Program Status Register reflect the results of this operation.

Overflow occurs if the original two doubleword operands had the same sign and the sign of the result was different.

DSUB Doubleword Subtract — Op Code: "4F01"

The doubleword operand in TOS and TOS1 is subtracted from the doubleword operand in TOS2 and TOS3. Both doubleword operands are popped from the stack. The doubleword difference is pushed into the stack. The carry and overflow bits of the Program Status Register reflect the results of this operation.

Overflow occurs if the original two doubleword operands had different signs and the sign of the result was not the same as the sign of the original second doubleword in the stack.

DNEG Doubleword Negate — Op Code: "3C"

The doubleword operand in TOS and TOS1 is replaced by its two's complement.

DABS Doubleword Absolute Value — Op Code: "3E"

If the doubleword operand in TOS and TOS1 is negative, it is replaced by its two's complement. If it is positive, it is left unchanged.

DMUL Doubleword Integer Multiply — Op Code: "4F02"

The doubleword operand in TOS2 and TOS3 is multiplied by the doubleword operand in TOS and TOS1. Both doubleword operands are popped, and the least significant 32 bits of their integer product are pushed into the stack.

DDIV Doubleword Integer Divide — Op Code: "4F03"

The doubleword operand in TOS and TOS1 is divided into the doubleword operand in TOS2 and TOS3. Both doubleword operands are popped from the stack. The doubleword integer quotient is pushed into the stack.

DIVD Divide Doubleword by Word — Op Code: "4F14"

The word operand in TOS is divided into the doubleword operand in TOS1 and TOS2. The word and the doubleword operand are popped. The 16-bit quotient is pushed into the stack.

DMOD Doubleword Modulo Op Code: "4F04"

The doubleword operand in TOS and TOS1 is divided into the doubleword operand in TOS2 and TOS3. Both doubleword operands are popped from the stack. The doubleword integer remainder is pushed into the stack.

MODD Doubleword Modulo by Word Op Code: "4F15"

The word operand in TOS is divided into the doubleword operand in TOS1 and TOS2. The word and doubleword operands are popped from the stack. The word integer remainder is pushed into the stack.

7.4 ARITHMETIC INSTRUCTIONS, FLOATING POINT OPERAND

Five floating point instructions are defined for arithmetic operations on floating point tripleword operands.

In all floating point operations the two operands are in TOS, TOS1, TOS2 and in TOS3, TOS4, and TOS5. The result is left in TOS, TOS1, TOS2. The operands are popped from the stack. The table below shows the operand locations:

OPERATION	TOS, TOS1, TOS2	TOS3, TOS4, TOS5
FADD	addend	augend
FSUB	subtrahend	minuend
FMUL	multiplier	multiplicand
FDIV	divisor	dividend

In all floating point operations, if either operand is an undefined floating point number, the result will contain all one bits.

FADD Floating Point Add — Op Code: "4F20"

The two tripleword operands are popped from the stack. Their floating point sum is pushed into the stack.

If the difference between the exponents is greater than 5 (in absolute value) the operand with the larger exponent is the result.

If the exponent difference is not greater than 5, addition takes place. The number with the smaller exponent is shifted right by the number of digits corresponding to the exponent difference. The exponent of the sum is the larger exponent. A zero digit is appended to the right of the larger number. Five digits of the smaller number are retained. The two fractions are then added. If the sum overflows, the intermediate sum is shifted right one digit; the most significant digit is set to one, and the exponent is increased by one. If an exponent overflow occurs, the result is set to all one bits in the most significant word.

After addition, the intermediate sum is shifted left until a non-zero digit is shifted into the most significant digit. The exponent is decreased by one for each position shifted. If an exponent underflow occurs, the result is a normal floating zero.

FSUB Floating Point Subtract — Op Code: "4F21"

The sign of the subtrahend is changed and the operation proceeds as in the FADD instruction.

FMUL Floating Point Multiply — Op Code: "4F22"

The two tripleword operands are popped from the stack. The normalized product of the operands is pushed into the stack.

The floating multiplication consists of characteristic addition and fraction multiplication. The characteristic of the product is the sum of the operand characteristics, less 64. The fractions are multiplied to form an 8-digit product. After multiplication, the product is normalized by shifting left until a non-zero digit is shifted into the most significant digit position. The exponent is decreased by one for each position shifted. If the resulting exponent underflows, the result is a normal zero. If the characteristic is greater than 127 after normalization, the result is set to all one's in the most significant word. The four most significant digits (after normalization) of the product are retained.

FDIV Floating Point Divide Op Code: "4F23"

The two tripleword operands are popped from the stack. The normalized quotient of the operands is pushed into the stack.

The division consists of a characteristic subtraction and a fraction division. The characteristic of the quotient is the difference between the dividend and divisor characteristics, plus 64. If the dividend fraction is greater than the divisor fraction, the dividend is shifted right 1 digit and the intermediate characteristic is reduced by 1. The resulting dividend fraction is divided by the divisor to form the quotient. If the divisor is zero, or if the characteristic overflows, the result is set to all one's. If the dividend is zero, or if the characteristic underflows, the result is a normal zero.

FABS Floating Point Absolute Value Op Code: "4F18"

If the tripleword floating point operand in the top of the stack is negative, it is set to positive. If the operand is positive, it is left unchanged.

7.5 MAXIMUM, MINIMUM AND SIGN VALUE INSTRUCTIONS

Six instruction types are provided to determine the maximum value and minimum value of pairs of word, doubleword, and floating point operands. Three instructions are provided to determine the sign of each of these types of operands.

MAX Maximum Value — Op Code: "34"

The two words on the top of the stack are popped and compared. The operand of greater arithmetic value is pushed into the stack.

MIN Minimum Value — Op Code: "35"

The two words on the top of the stack are popped and compared. The operand of lesser arithmetic value is pushed into the stack.

DMAX Doubleword Maximum Value — Op Code: "4F0E"

The two doublewords on the top of the stack are popped and compared. The operand of greater arithmetic value is pushed into the stack.

DMIN Doubleword Minimum Value — Op Code: "4F0F"

The two doublewords on the top of the stack are popped and compared. The operand of lesser arithmetic value is pushed into the stack.

FMAX Floating Point Maximum Value — Op Code: "4F2E"

The two floating point operands on the top of the stack are popped and compared. The operand of greater arithmetic value is pushed into the stack. If either or both are undefined, the result will be set to all one bits.

FMIN Floating Point Minimum Value — Op Code: "4F2F"

The two floating point operands on the top of the stack are popped and compared. The operand of lesser arithmetic value is pushed into the stack. If either or both are undefined a word of the result will be set to all one bits.

SGN Sign Value — Op Code: "44"

The word on the top of the stack is popped and tested. If the operand is zero, a zero is pushed into the stack. Otherwise a '1' value is pushed into the stack with the sign of the operand.

DSGN Doubleword Sign Value — Op Code: "45"

The doubleword on the top of the stack is popped and tested. If the operand is zero, a doubleword zero is pushed into the stack. Otherwise a doubleword '1' is pushed into the stack with the sign of the operand.

FSGN Floating Point Sign Value — Op Code "4F19"

The floating point operand on the top of the stack is popped and tested. If the operand is zero, a floating point zero is pushed into the stack. Otherwise a floating point '1' is pushed into the stack with the sign of the operand.

7.6 LOGICAL INSTRUCTIONS

Eight stack operate logical instructions are defined, four with word operands and four with doubleword operands.

AND Logical AND — Op Code: "25"

Logically AND the two words on the top of the stack and replace the second word with the result. Then pop the top of the stack.

NOT Logical Not — Op Code: "11"

Replace the word on the top of the stack by its one's complement.

OR Logical OR — Op Code: "26"

Logically OR the two words on the top of the stack and replace the second word with the result. Then pop the top of the stack.

XOR Logical EXCLUSIVE OR — Op Code: "27"

Logically EXCLUSIVE OR the two words on the top of the stack and replace the second word with the result. Then pop the stack.

DAND Doubleword Logical AND — Op Code: "4F05"

Logically AND the two doublewords on the top of the stack and replace the second doubleword with the result. Then pop the doubleword from the top of the stack.

DNOT Doubleword Logical Not — Op Code: "3D"

Replace the doubleword on the top of the stack with its one's complement.

DOR Doubleword Logical OR — Op Code: "4F06"

Logically OR the two doublewords on the top of the stack and replace the second doubleword with the result. Then pop the doubleword from the top of the stack.

DXOR Doubleword Logical EXCLUSIVE OR — Op Code: "4F07"

Logically EXCLUSIVE OR the two doublewords on the top of the stack and replace the second doubleword with the result. Then pop the doubleword from the top of the stack.

7.7 COMPARISON INSTRUCTIONS

Twenty-two stack operate comparison instructions are defined: 10 for word operands, six for doubleword operands, and six for floating point (tripleword) operands.

All stack operate comparison instructions perform similar operations:

1. The two operands are popped from the top of the stack.
2. The operand which had been second in the stack is compared to the operand which had been in the top of the stack on the basis specified by the instruction name; e.g., second operand greater than first operand.
3. If the result of the comparison is true, a word with the value of 1 is pushed into the stack. If the result of the comparison is false, a word with the value of 0 is pushed into the stack.
4. If the comparison is being performed on floating point (tripleword) operands, and if either or both of the operands are undefined, the comparison cannot be meaningfully performed. In this case, a word with the value of 2 is pushed into the stack.

In the instruction definitions which follow, only the basis for a true comparison result is given.

EQ	Equal Comparison	Op Code: "2A"
	Word in TOS1 equal to word in TOS.	
GE	Greater Than or Equal Comparison	Op Code: "2C"
	Word in TOS1 arithmetically greater or equal to word in TOS.	
LGE	Logical Greater Than or Equal Comparison	Op Code: "3A"
	Word in TOS1 greater or equal to word in TOS, both words treated as 16 bit positive numbers.	
GT	Greater Than Comparison	Op Code: "2D"
	Word in TOS1 arithmetically greater than word in TOS1.	
LGT	Logical Greater Than Comparison	Op Code: "3B"
	Word in TOS1 greater than word in TOS1, both words treated as 16 bit positive numbers.	
LE	Less Than or Equal Comparison	Op Code: "29"
	Word in TOS1 arithmetically less than or equal to word in TOS.	
LLE	Logical Less Than or Equal Comparison	Op Code: "39"
	Word in TOS1 less than or equal to word in TOS, both words treated as 16 bit positive numbers.	
LT	Less Than Comparison	Op Code: "28"
	Word in TOS1 arithmetically less than word in TOS.	

LLT	Logical Less Than Comparison	Op Code: "38"
	Word in TOS1 less than word in TOS, both words treated as 16-bit positive numbers.	
NE	Not Equal Comparison	Op Code: "2B"
	Word in TOS1 not equal to word in TOS.	
DEQ	Doubleword Equal Comparison	Op Code: "4F0A"
	Doubleword in TOS2 and TOS3 equal to doubleword in TOS and TOS1.	
DGE	Doubleword Greater Than or Equal Comparison	Op Code: "4F0C"
	Doubleword in TOS2 and TOS3 arithmetically greater than or equal to doubleword in TOS and TOS1.	
DGT	Doubleword Greater Than Comparison	Op Code: "4F0D"
	Doubleword in TOS2 and TOS3 arithmetically greater than doubleword in TOS and TOS1.	
DLE	Doubleword Less Than or Equal Comparison	Op Code: "4F09"
	Doubleword in TOS2 and TOS3 arithmetically less than or equal to doubleword in TOS and TOS1.	
DLT	Doubleword Less Than Comparison	Op Code: "4F08"
	Doubleword in TOS2 and TOS3 arithmetically less than doubleword in TOS and TOS1.	
DNE	Doubleword Not Equal Comparison	Op Code: "4F0B"
	Doubleword in TOS2 and TOS3 not equal to doubleword in TOS and TOS1.	
FEQ	Floating Point Equal Comparison	Op Code: "4F2A"
	Floating point number in TOS3, TOS4, and TOS5 equal to floating point number in TOS, TOS1 and TOS2.	
FGE	Floating Point Greater Than or Equal Comparison	Op Code: "4F2C"
	Floating point number in TOS3, TOS4, and TOS5 arithmetically greater than or equal to floating point number in TOS, TOS1, and TOS2.	
FGT	Floating Point Greater Than Comparison	Op Code: "4F2D"
	Floating point number in TOS3, TOS4, and TOS5 arithmetically greater than floating point number in TOS, TOS1 and TOS2.	
FLE	Floating Point Less Than or Equal Comparison	Op Code: "4F29"
	Floating point number in TOS3, TOS4, and TOS5 arithmetically less than or equal to floating point number in TOS, TOS1, and TOS2.	
FLT	Floating Point Less Than Comparison	Op Code: "4F28"
	Floating point number in TOS3, TOS4, and TOS5 arithmetically less than floating point number in TOS, TOS1, and TOS2.	
FNE	Floating Point Not Equal Comparison	Op Code: "4F2B"
	Floating point number in TOS3, TOS4, and TOS5 not equal to floating point in TOS, TOS1, and TOS2.	

7.8 SHIFT INSTRUCTIONS

Eight stack operate shift instructions are defined; four with word operands and four with doubleword operands.

SLC Shift Left Circular — Op Code: "33"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to circularly left shift the word now on top of the stack. Bits shifted off of the most significant end of the word are inserted into the vacated least significant bits.

The shift count is treated as modulo 16.

SLL Shift Left Logical — Op Code: "30"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to left shift the word now on the top of the stack. Zeroes are inserted into vacated bit positions. Bits shifted off the most significant end of the word are lost.

The shift count is treated as modulo 16.

SRA Shift Right Arithmetic — Op Code: "31"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to right shift the word now on the top of the stack. The sign bit is extended to the right. Bits shifted off the least significant end of the word are lost.

The shift count is treated as modulo 16.

SRL Shift Right Logical — Op Code: "32"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to right shift the word now on the top of the stack. Zeroes are inserted into the vacated most significant bit. Bits shifted off the least significant end of the word are lost.

The shift count is treated as modulo 16.

DSLC Doubleword Shift Left Circular — Op Code: "4F13"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to circularly left shift the doubleword now on top of the stack. Bits shifted off of the most significant end of the doubleword are inserted into the vacated least significant bits.

The shift count will be treated as modulo 32.

DSLL Doubleword Shift Left Logical — Op Code: "4F10"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to left shift the doubleword now on the top of the stack. Zeroes are inserted into the vacated least significant bits. Bits shifted off of the most significant end are lost.

The shift count is treated as modulo 32.

DSRA Doubleword Shift Right Arithmetic Op Code: "4F11"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to right shift the doubleword now on the top of the stack. The sign bit is extended to the right. Bits shifted off the least significant end of the doubleword are lost.

The shift count is treated as modulo 32.

DSRL Doubleword Shift Right Logical Op Code: "4F12"

A shift count word is popped from the top of the stack and is used to determine the number of bit positions to right shift the doubleword now on the top of the stack. Zeroes are inserted into the vacated most significant bits. Bits shifted off the least significant end of the word are lost.

The shift count is treated as modulo 32.

7.9 LOAD LITERAL & ENTER CONFIGURATION SWITCH INSTRUCTIONS

Stack operate instructions are defined to push 4-bit, byte, word, doubleword, and variable word-length literals into the stack. An additional instruction is defined to sense and push the state of the configuration switches into the stack.

Ln Load 4 Bit Literal — format: "7n"

The lower four bits (n) of the instruction are pushed onto the top of the stack as an integer word with a value of 0-15.

The complete set of instruction mnemonics is: L0, L1, L2, L3, L4, L5, L6, L7, L8, L9, L10, L11, L12, L13, L14, L15.

LBL Load Byte Literal — format: "41 xx"

The 8-bit byte literal contained in the second byte of the instruction (xx) is pushed onto the top of the stack as a 16-bit word. The upper 8-bits of the word on the top of the stack will be set to all zeroes.

LWL Load Word Literal — format: "40 xx xx"

The word length literal contained in the second and third bytes of this instruction (xxxx) is pushed onto the top of the stack.

LDL Load Doubleword Literal — format: "42 xx xx xx xx"

The doubleword length literal contained in the second, third, fourth, and fifth bytes of the instruction (xxxxxxxx) is pushed onto the top of the stack. The least significant half of the doubleword will be TOS; the most significant half in TOS1.

LTL Load Tripleword Literal — format: "43 xx xx xx xx xx xx"

The tripleword length literal contained in the second through seventh bytes of this instruction (xxxxxxxxxxxx) is pushed onto the top of the stack. The most significant word will be in TOS2 and the least significant word in TOS.

FILL Fill Stack with Literal — format: "59 yy xx xx ...."

The second byte of the instruction (yy) is a word count. The number of words specified by this count is pushed into the stack. The stack is stuffed into memory.

ESW Enter Configuration Switches — format: "05"

The conditions of the four internal configuration switches on the processor data board are copied into the low order four bits of a word which is pushed onto the top of the stack. The upper 12 bits of the word are zeroes. These four switches are the same switches used during an Initial Program Load to select the load device.

7.10 STACK MODIFY INSTRUCTIONS

Eight stack operate instructions are defined to modify the contents of the top of the stack.

Eight instructions are provided to modify the contents of the top of the stack. 'Before and after' pictures of the data stack are shown in Figure A for six of these instructions.

DBL1 Convert TOS to Doubleword — Op Code: "08"

The signed word operand in TOS is popped from the stack and converted into a signed doubleword operand. This doubleword operand is pushed into the stack. The most significant half of the doubleword result (in TOS1) is the sign bit extension (either all zeroes or all one's) of the original word operand. The least significant half of the doubleword result (in TOS) is identical to the original word operand.

DBL2 Convert TOS2 to Doubleword — Op Code: 4F16"

The words in TOS and TOS1 are popped and saved. An operation identical to execution of the DBL1 instruction is performed. The two words originally in TOS and TOS1 are then pushed back into the stack.

SNGL Convert Doubleword to Single Word — Op Code: "37"

The signed doubleword operand in TOS and TOS1 is popped and converted to a signed word operand. The resultant word is pushed into the stack. The resultant word (in TOS) is identical to the least significant half of the original doubleword operand.

Note: Overflow occurs unless, if in the original doubleword, the most significant 17 bits are either all one's or all zeroes.

DDUP Duplicate Doubleword on Top of Stack — Op Code: "3F"

The doubleword operand in the top of the stack is duplicated by pushing a copy of that doubleword into the stack.

DUP Duplicate Top of Stack — Op Code: "1F"

The word operand in the top of the stack is duplicated by pushing a copy of that word into the stack.

XCH Exchange Top of Stack Words — Op Code: "2F"

The two word operands in the top of the stack are exchanged.

FLOT Float an Integer — Op Code: "4C"

The doubleword integer operand in TOS and TOS1 is popped from the stack and converted into a floating point tripleword operand. The resultant tripleword is pushed into the stack.

FIX Fix a Floating Point Number — Op Code: "4F17"

The tripleword floating point operand in TOS, TOS1, and TOS2 is popped from the stack and converted into a doubleword integer operand. The resultant doubleword is pushed into the stack.

The fractional part of the floating point number is lost in the conversion process. If the number is greater than 2,147,483,647 in absolute value, the overflow indicator is set in the Program Status Register and the resultant doubleword (in TOS and TOS1) is not predictable.

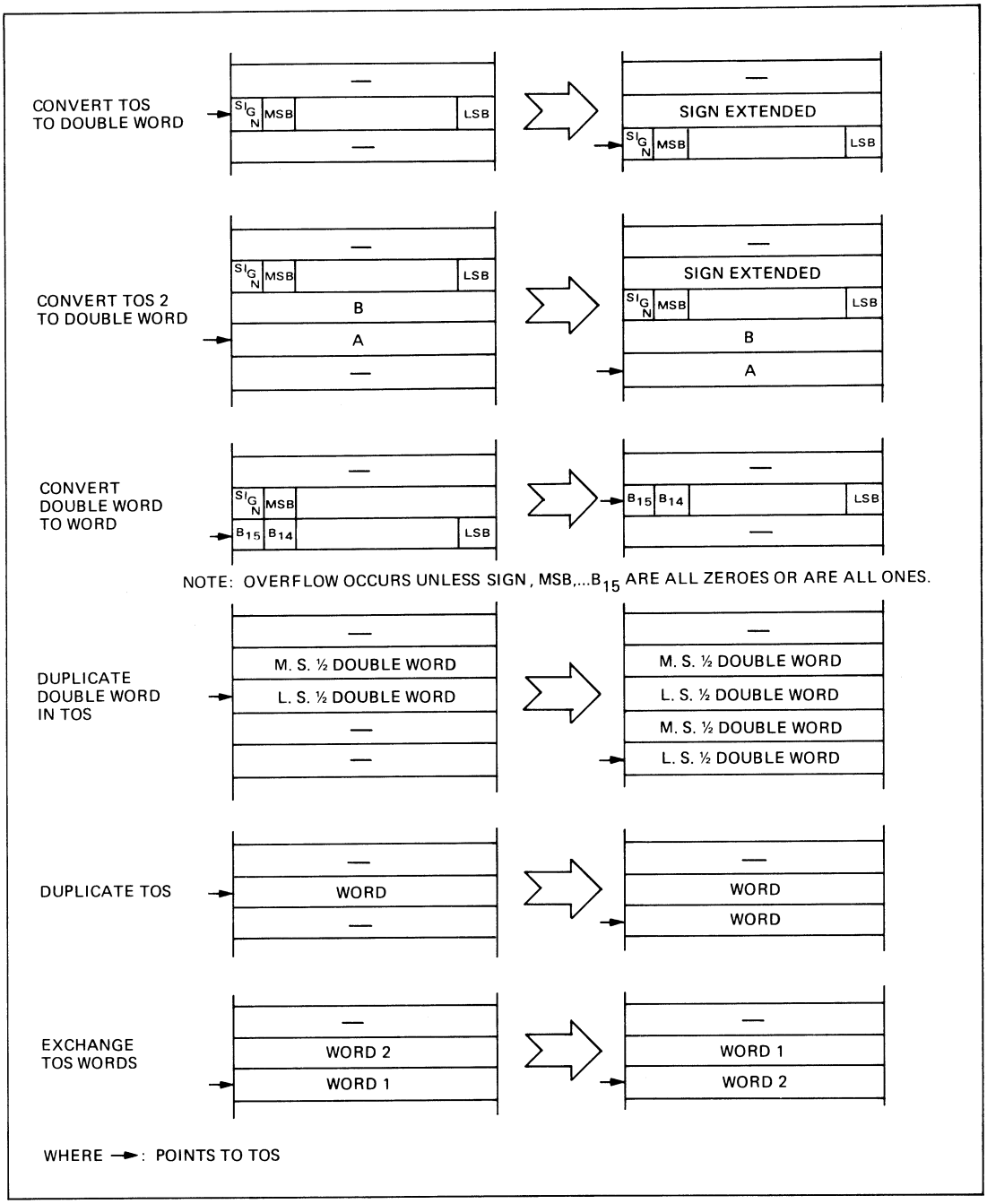


Figure A. Memory Before and After Execution of Stack Modification Instructions.

7.11 FIELD DESCRIPTOR GENERATION INSTRUCTION

An instruction is provided to generate field descriptors in the top of the stack.

The field descriptor is used to extract a field from a word in memory and to store a field into a word in memory. The explanation of its use is given in topic 5.1

The format of the field descriptor is shown in Figure A. The rightmost field specifies the location of the field within the word. The second rightmost field specifies the field length. Specifically:

LSBP (bits 3-0): binary-encoded bit position of the least significant bit position of the field within the word.

FL-1 (bits 7-4): binary-encoded number of bit positions, minus one, in the field.

Before and after pictures of the stack are shown in Figure B for the Generate Field Descriptor instruction.

GFD Generate Field Descriptor Op Code: "2E"

This instruction converts the two words on the top of the stack into an appropriate field descriptor word for the load field (LF) and store field (SF) memory reference instructions.

The low order four bits of the second word on the stack are decremented by one, shifted left four bit positions, and logically ORed with the low order four bits of the word on the top of the stack. The result replaces the second word on the stack. The top of the stack is popped.

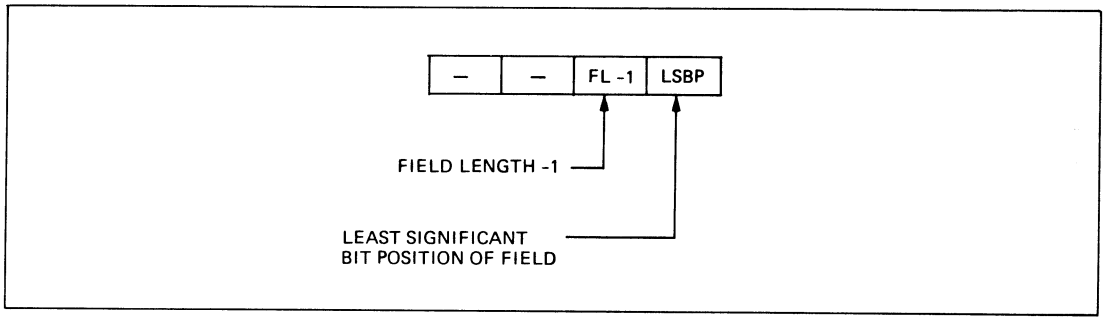


Figure A. Field Descriptor Format.

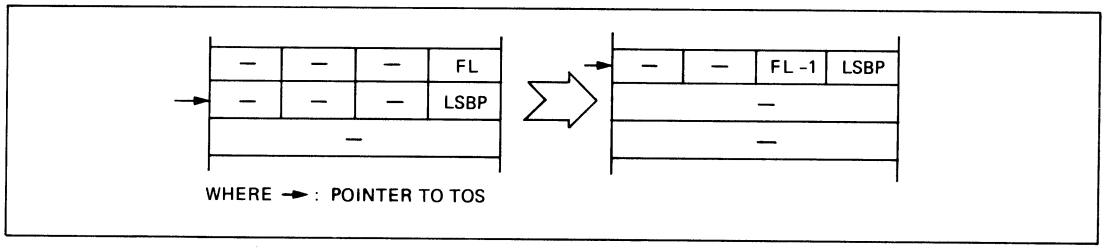


Figure B. Memory Before and After Execution of Generate Field Descriptor Instruction.

7.12 BIT ARRAY INSTRUCTIONS

Three special instructions are provided to convert a bit array index to a word index for use by indexed Load Field and Store Field instructions.

BIT(1), BIT(2), and BIT(4) data types are defined in MPL. Declaration of these data types reserves arrays of one-bit, two-bit, and four-bit fields in memory. The fields are specified to begin at Monobus word location boundaries. Arrays of these fields are indexed by an "array index." See Figure A.

The Load Field and Store Field memory reference instructions are used to load and store individual fields of these arrays. To index into the arrays these instructions utilize the indexed addressing modes (modes 3, 5, 6 and 7). This mode of addressing (for the Load Field and Store Field) requires that a word index and a field descriptor be in the top of the stack before instruction execution begins. (See topics 6.3 and 6.4.)

Three types of instructions are provided to generate a word index and a field descriptor from an array index. The relationship between an array index and the corresponding word index and field descriptor is illustrated in Figure B.

The three types of instructions correspond to the three types of bit arrays, BIT(1), BIT(2), and BIT(4). These instructions take an array index in TOS and convert it to a word index in TOS1 and a field descriptor in TOS. The field descriptor has an appropriate Field Length -1 value for the field size specified (0, 1, or 3). See Figure C.

XB1 Convert Index for BIT(1) Arrays — Op Code: "0D"

Pop the Bit(1) array index from the top of the stack and convert it into a word index which is pushed onto the stack and an appropriate field descriptor word which is also pushed onto the stack. This field descriptor and index will be used by a subsequent load field or store field instruction to access the appropriate element in an array of one bit items.

XB2 Convert Index for BIT(2) Arrays — Op Code: "0E"

Pop the Bit(2) array index from the top of the stack and convert it into a word index which is pushed onto the stack and an appropriate field descriptor word which is also pushed onto the stack. This field descriptor and index will be used by a subsequent load field or store field instruction to access the appropriate element in an array of two bit items.

XB4 Convert Index for BIT(4) Arrays — Op Code: "0F"

Pop the Bit(4) array index from the top of the stack and convert it into a word index which is pushed onto the stack and an appropriate field descriptor word which is also pushed onto the stack. This field descriptor and index will be used by a subsequent load field or store field instruction to access the appropriate element in an array of four-bit terms.

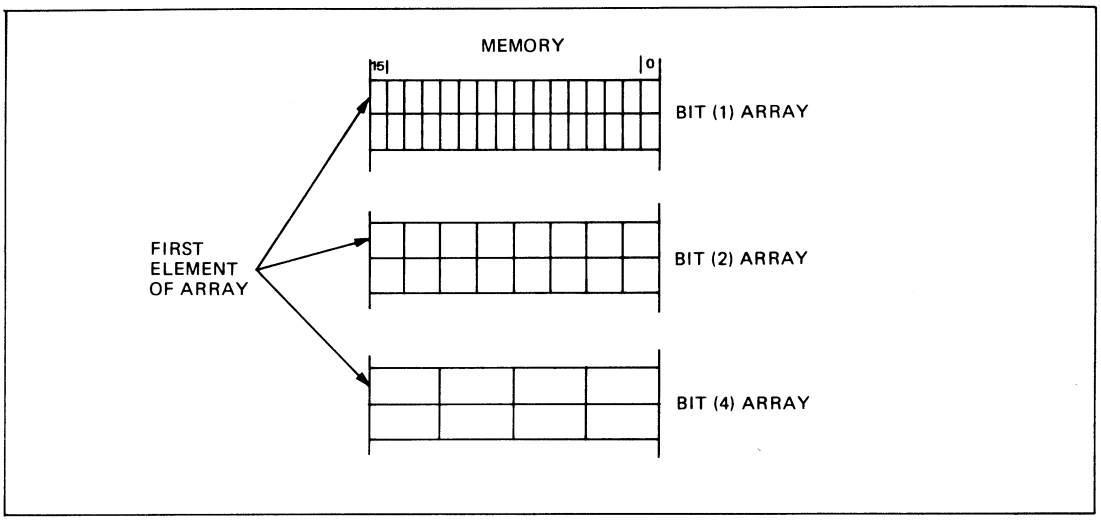


Figure A. Bit (n) Array Formats.

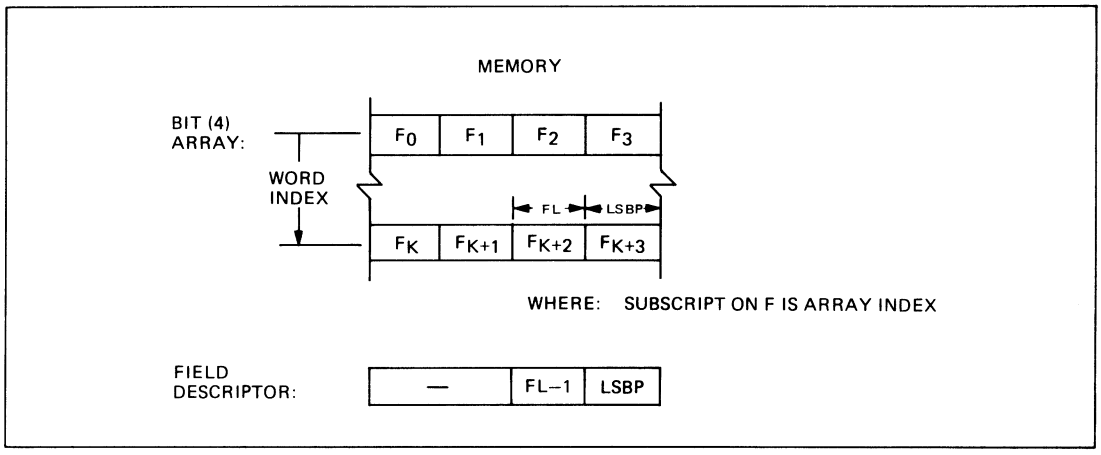


Figure B. Relationship Between Word Index/Field Descriptor and Array Index.

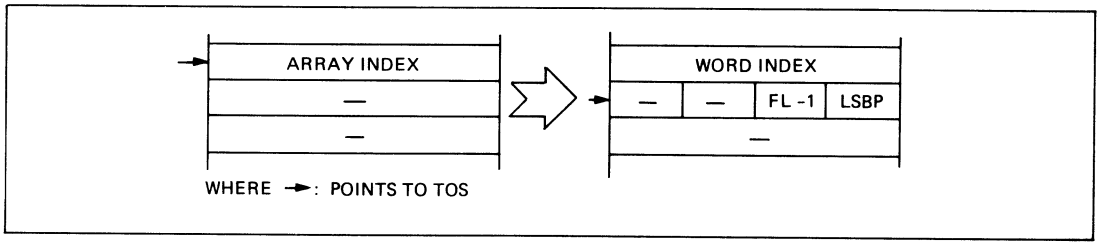


Figure C. Stack Before and After Execution of Convert Index for Bit (n) Array.

8 Branch Instructions

8.1 BRANCH INSTRUCTIONS - INTRODUCTION

Seventeen types of branch instructions are provided. Four of these are specialized to the functions specified by MPL DO statements.

Three categories of simple branch instructions are provided: a relative branch backwards from the current Program Pointer value; a direct branch; and an indirect branch via the contents of TOS. Both conditional and unconditional branch instructions are provided within each category (except the indirect, which is only unconditional). The three categories of simple branch instructions are shown in Figure A.

Branch backward instructions use a two byte format, with the rightmost byte being an eight bit displacement, D8. The effective address to be placed in the Program Pointer, PP, is computed by subtracting the displacement D8 from the current value of PP. At the time that the effective address is computed, PP is pointing to the displacement byte of the instruction (therefore a D8 value of one would specify a branch back to the instruction itself). Both conditional and unconditional branch backward instructions are provided.

Branch long instructions use a three-byte format, with the rightmost two bytes being a 16-bit direct address, ADDR, to be placed in the Program Pointer, PP. Both unconditional and conditional (based upon the contents of TOS) branch long instructions are provided.

Branch indirect via TOS instructions use a one-byte format. The two-byte indirect address is provided in TOS. An unconditional branch instruction is provided.

Four specialized branch instructions are provided to facilitate the compilation of MPL DO statements. These instructions are shown in Figure B.

The CASE branch instruction provides a multiway indirect branch. It uses a three-byte format, with the rightmost two bytes being a 16-bit address, ADDR. The effective address to be placed in the Program Pointer is retrieved from an indirect address table which begins at the address: Program Base (PB) plus ADDR. The entry within the indirect address table is specified by an index value which is in the top of the stack.

The DO Loop Initialize and Branch instruction initializes the data stack for performing the DO loop code, tests the initial and final variable values to see if loop execution should be performed and, if not, branches the program around the DO loop code. It uses a three-byte format, with the rightmost two bytes, ADDR, being the effective address to be placed in the Program Pointer if the DO loop is not to be executed.

The DO Loop Step, Branch Backward and DO Loop Step, Branch Long instructions test the current and final variable values to see if another iteration of the DO loop code is to be done and, if it is, branch the program to the start of the DO loop code. The branch backward instruction uses a two-byte format, with the rightmost byte being eight-bit displacement, D8. The effective address to be placed in the Program Pointer (if the DO loop is to be executed) is computed by subtracting the displacement D8 from the current value of PP. The branch long instruction uses a three-byte format, with the rightmost two bytes being a 16-bit direct address, ADDR, to be placed in the Program Pointer (if the DO loop is to be executed).

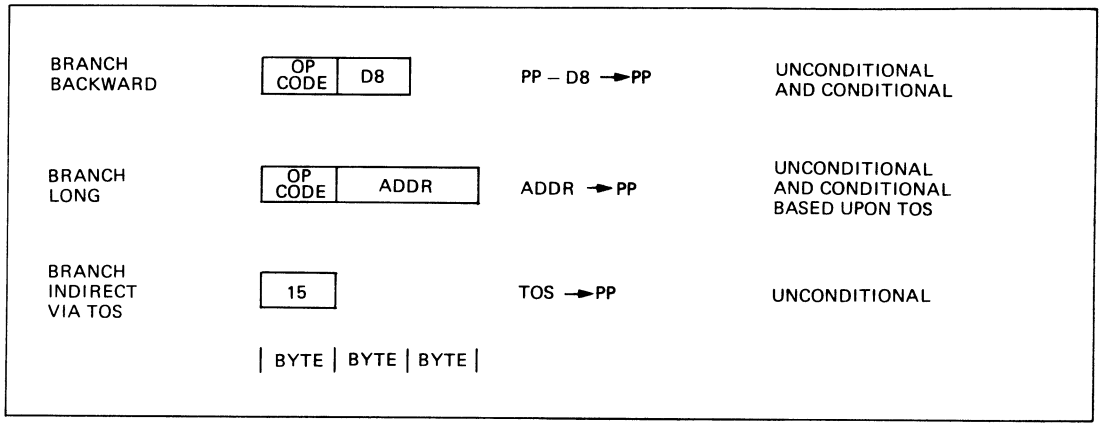


Figure A. Simple Branch Instructions.

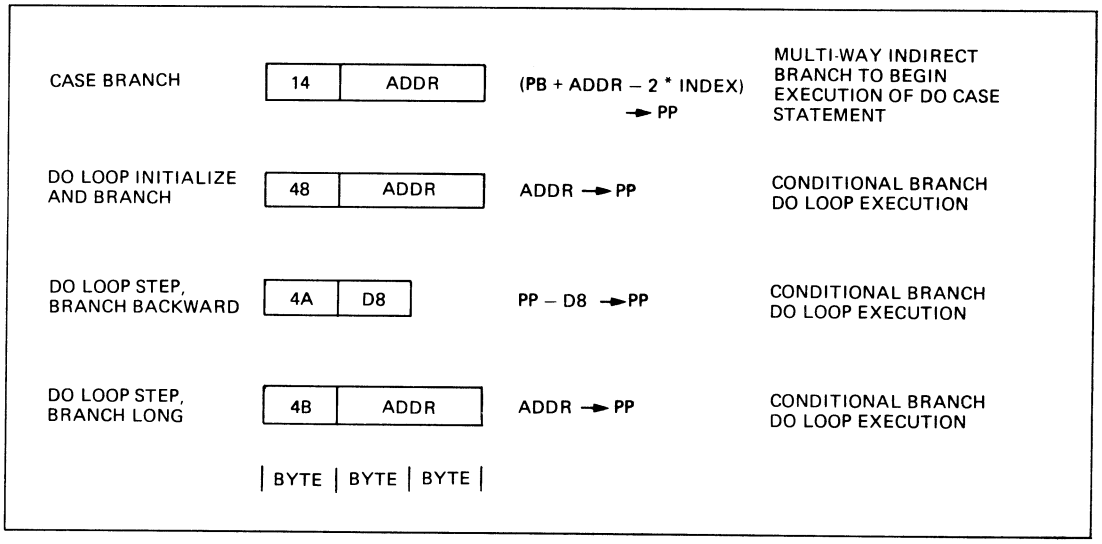


Figure B. Specialized Branch Instructions.

8.2 SIMPLE BRANCH INSTRUCTIONS

Three types of unconditional branch instructions, one with each type of branch addressing (relative backwards, direct, and indirect via TOS) are defined. Ten types of conditional branch instructions, all but one with direct addressing, are defined.

The simple branch instructions utilize the three formats and the three corresponding effective address expressions described in topic 8.1.

BRB Branch Backward format: "46 xx"

Branch backward across the number of bytes specified by the second byte (xx) of the instruction.

BRA Branch format: "47 xx xx"

Branch to the PB relative address contained in the second and third bytes of the instruction (xxxx).

BTOS Branch Through Top of Stack format: "15"

Branch to the PB relative address contained in the word on the top of the stack. Pop the word from the top of the stack.

DBB Decrement TOS and Branch Backward format "16 xx"

Decrement the word on the top of the stack and if the result is non-negative, branch backward across the number of bytes specified by the second byte (xx) of the instruction.

If the result was negative, pop the value from the top of the stack and execute the next instruction in sequence.

BEQZ Branch If TOS Equal to Zero format: "1A xx xx"

Pop the word from the top of the stack and if it is equal to zero, branch to the PB relative address contained in the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

BGEZ Branch If TOS Greater Than or Equal to Zero format: "1C xx xx"

Pop the word from the top of the stack and if it is greater than or equal to zero, branch to the PB relative address contained in the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

DBL Decrement TOS and Branch Long format: "17 xx xx"

Decrement the word on the top of the stack and if the value is non-negative, branch to the address contained in the second and third bytes of the instruction (xxxx).

If the result was negative, pop the word from the top of the stack and execute the next instruction in sequence.

BGTZ Branch If TOS Greater Than Zero format: "1D xx xx"

Pop the word from the top of the stack and if it is greater than zero, branch to the PB relative address contained in the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

BLEZ Branch If TOS Less Than Or Equal to Zero format: "19 xx xx"

Pop the word from the top of the stack and if it is less than or equal to zero, branch to the PB relative address contained in the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

BLTZ Branch If TOS Less Than Zero format: "18 xx xx"

Pop the word from the top of the stack and if it is less than zero, branch to the PB relative address contained in the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

BNEZ Branch If TOS Not Equal to Zero format: "1B xx xx"

Pop the word from the top of the stack and if it is not equal to zero, branch to the PB relative address contained in the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

BRF Branch False format: "13 xx xx"

Pop the word from the top of the stack and if the least significant bit of the word is zero, branch to the PB relative location contained in the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

BRT Branch True format: "12 xx xx"

Pop the word from the top of the stack and if the least significant bit of the word is a one, branch to the PB relative address specified by the second and third bytes of this instruction (xxxx); otherwise execute the next instruction in sequence.

8.3 CASE BRANCH INSTRUCTION

The CASE branch instruction provides a multiway indirect address branch for compilation of DO CASE MPL statements.

The CASE branch instruction utilizes a three byte format and specifies the Program-Base relative address of an indirect address table of branch addresses which must be aligned on a word boundary. An index into this table is provided in the top of the stack. Figure A shows the instruction format, the location and indexing of the indirect address table in the program segment, and the before and after pictures of the data stack.

As indicated in Figure A, the indirect address table is indexed in an inverted order, and the index. which is provided to the instruction in TOS, is multiplied by two to provide a word-level index. The continuation point of the program, the next word after the indirect address table, is pushed into the stack after the index is popped. This address is normally used by a top of stack branch (BTOS) instruction as the program continuation point after executing the instruction for the selected case.

CASE CASE Branch format: "14 xx xx"

The PB relative address contained in the ADDR field of the instruction (xxxx), minus two times the value of the index word on the top of the stack, points to a branch address which is placed in the Program Pointer. The address in the instruction, ADDR, plus two, replaces the contents of the stack.

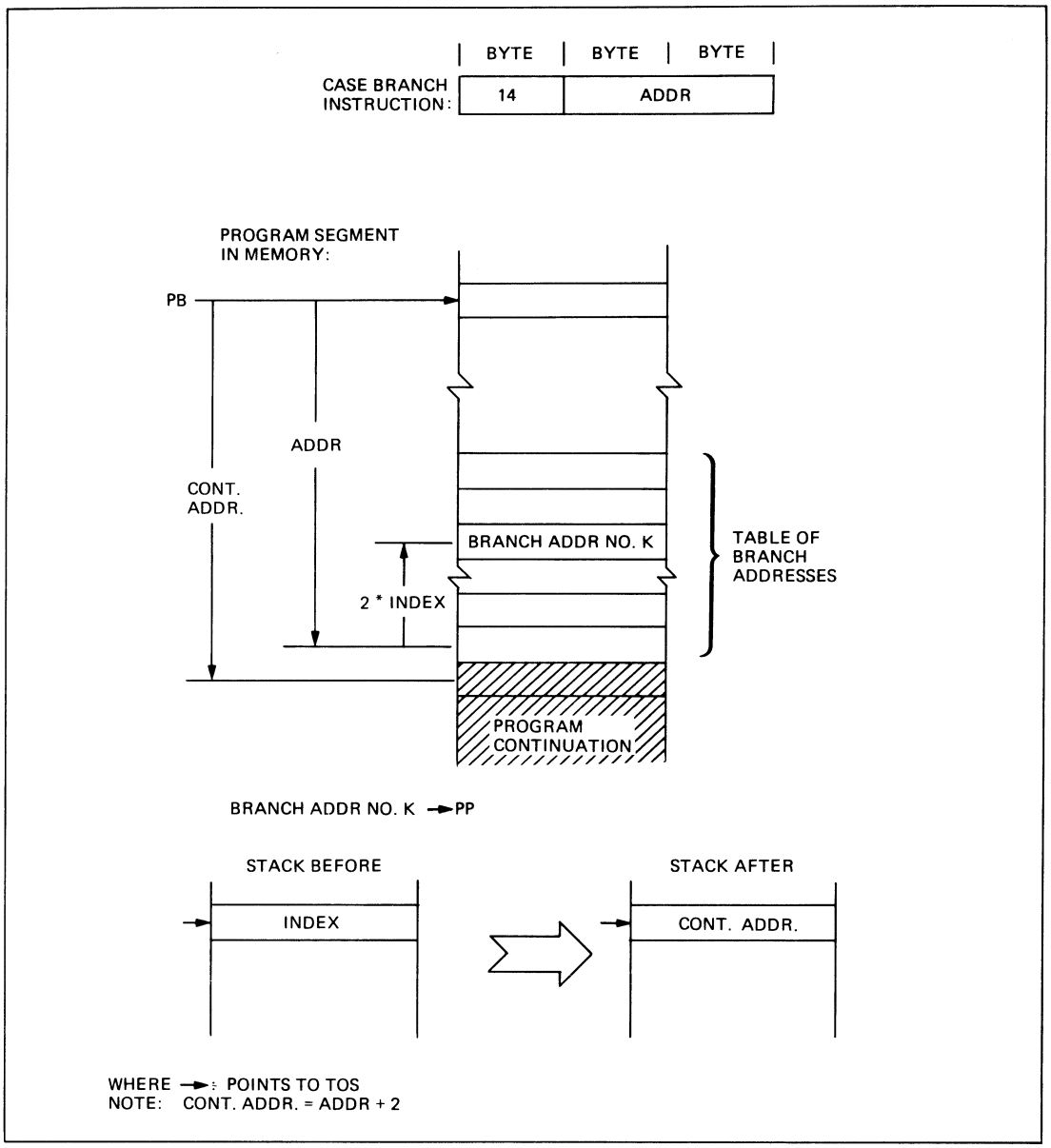


Figure A. The Case Branch Instruction.

8.4 DO LOOP INITIALIZE & BRANCH INSTRUCTION

The DO Loop Initialize and Branch instruction initializes the data stack for performing the DO loop code, tests the initial and final values of the variable to see if the loop should be executed and, if it should, branches to the start of the DO loop code.

The DO Loop Initialize and Branch instruction utilizes a three-byte instruction format and specifies the Program Base relative address to be placed in the Program Pointer if the DO loop is not to be executed. The initial, final, and step control parameter values for the DO statement are provided in the top of the stack, along with the Stack Base relative address into which the initial control parameter value is to be stored.

Figure A shows the instruction format. It also shows the before and after pictures of the data stack if the DO loop is not executed and the branch is taken; if the DO loop is executed, the stack remains as in the "before" picture. Figure A also shows a flowchart of the instruction execution.

DIB DO Loop Initialize and Branch format: "48 xx xx"

The Stack Head Registers are pushed into the stack in memory. The INITIAL value of the DO loop control variable contained in TOS3 is stored at the SB relative address contained in TOS.

The INITIAL value of the control variable is then compared with the FINAL value in TOS2 as directed by the sign of the STEP value in TOS1 to determine if a branch will occur.

If the sign of STEP is positive, and the value of INITIAL is less than or equal to the value of FINAL, or if the sign of STEP is negative and the value of INITIAL is greater than or equal to the value of FINAL, then the next instruction in sequence is executed. If the INITIAL value exceeds the FINAL value in the direction of the sign of STEP, then the four words on the top of the stack are popped and a branch is executed.

If the branch is to be executed, the ADDR field in the second and third bytes of the instruction (xxxx) specify the PB relative address of the branch.

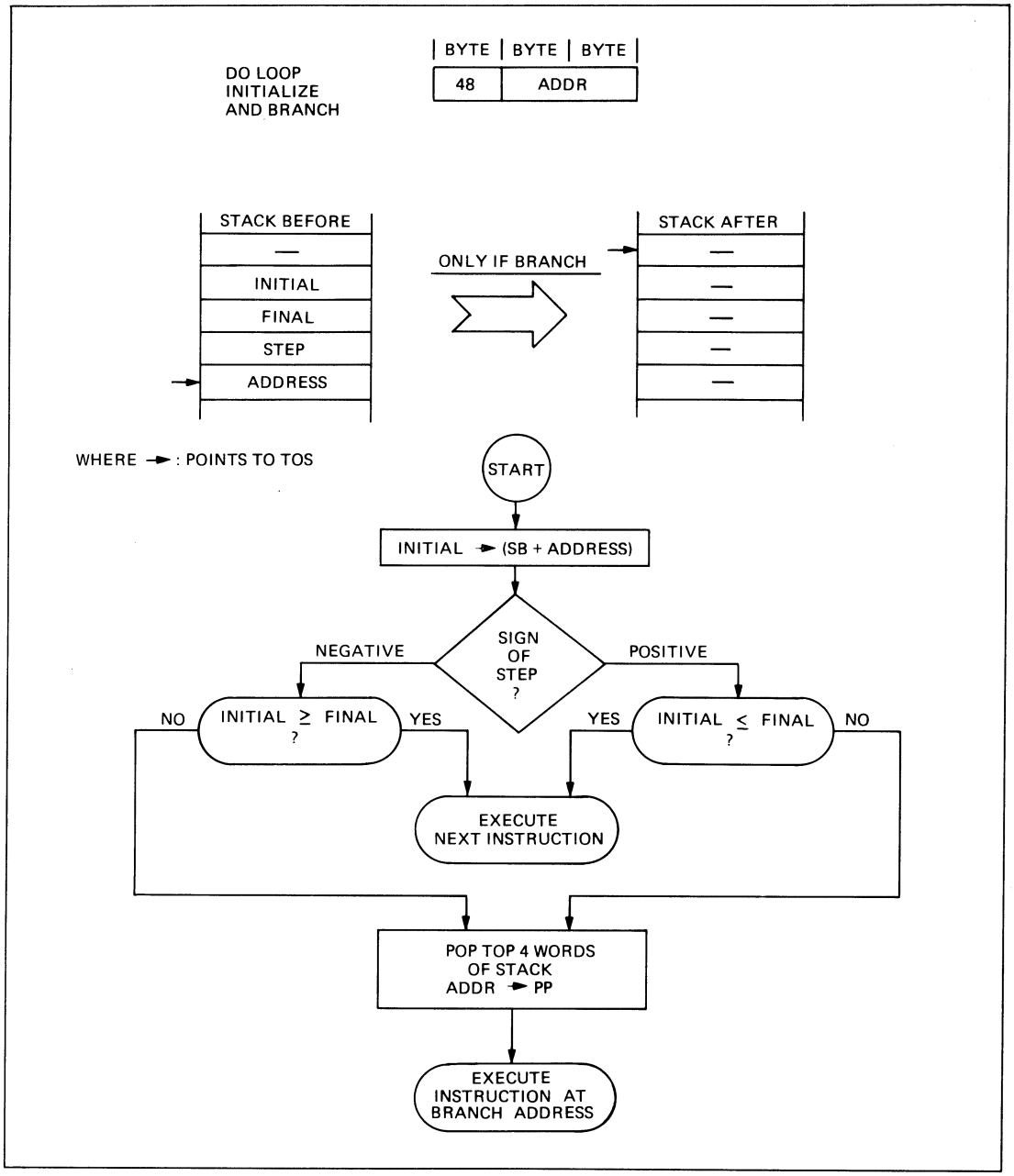


Figure A. De Loop Initialize and Branch Instruction.

8.5 DO LOOP STEP, BRANCH BACKWARD & BRANCH LONG INSTRUCTIONS

The two DO Loop Step instructions each determine if another iteration of the DO loop code is to be performed, and branch to that code if it is to be executed.

The DO Loop Step instructions utilize two different formats with corresponding effective address expressions. The DO Loop Step, Branch Backward instruction uses a two-byte format which provides an 8-bit displacement to be subtracted from the Program Pointer if the branch is taken. The DO Loop Step, Branch Long instruction uses a three-byte format which provides a 16-bit address to be placed into the Program Pointer if the branch is to be taken. The initial, final, and step control parameter values for the DO statement are provided in the top of the stack, along with the Stack Base-relative address of the current control parameter value.

Figure A shows the format of the two types of instructions. It also shows the before and after pictures of the data stack if the DO loop is not executed and the branch is not taken. If the DO loop is executed, the stack remains as in the "before" picture. Finally, Figure A shows a flow chart of the instruction execution.

DSBB DO Loop Step, Branch Backward format: "4A xx"

'The word in TOS is the SB relative address of the current control variable. The STEP value contained in TOS1 is added to this current control variable to obtain the new current value. If an arithmetic overflow occurs in this addition, the four words are popped and the branch is not taken. The Overflow indicator, however, is not affected by this instruction.

The new control variable value is then compared with the FINAL value in TOS2 as directed by the sign of the STEP value in TOS1 to determine if a branch will occur.

If the sign of STEP is positive and the value of the new control variable is greater than the value of FINAL, or if the sign of STEP is negative and the value of the new control variable is less than the value of FINAL, then the next instruction is executed and the top four stack values are popped. If the current control variable is less than or equal to the FINAL value in the direction of the sign of STEP, the stack is unaltered and the branch occurs.

If the branch occurs, the branch location is computed by subtracting the value of the D8 field of the instruction (xx) from the Program Pointer.

DSBL DO Loop Step, Branch Long format: "4B xx xx"

The operation of this instruction is the same as DSBB except for the branch address determination.

If the branch is to be executed, the Program Base relative address contained in the ADDR field of the instruction (xxxx) is placed in the Program Pointer.

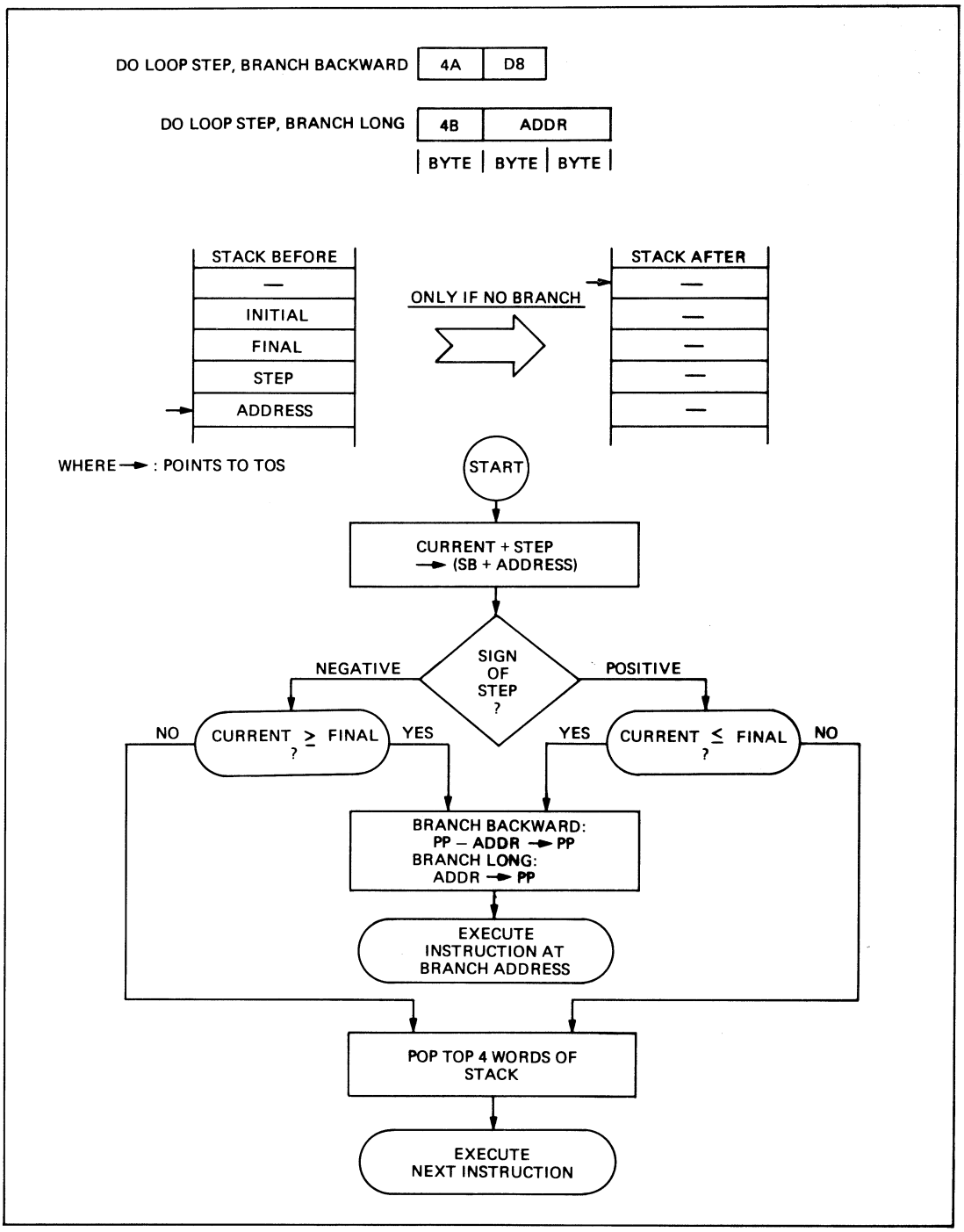


Figure A. Do Loop Step, Branch Backward and Branch Long Instructions.

9 Control Instructions

9.1 BEGIN BLOCK ENTRY & BEGIN BLOCK EXIT INSTRUCTIONS

The Begin Block Entry instruction creates a Begin Mark in the data stack and the Begin Block Exit instruction removes the Begin Mark.

The Begin Block Entry instruction is compiled for an MPL BEGIN statement. Its function is to create and activate the Begin Mark in the top of the data stack to delineate the start of the stack environment for the BEGIN block. The Begin Block Exit instruction is compiled for the MPL END statement which terminates the BEGIN block. Its function is to "roll back" the stack to the environment of the preceding block by resetting the Environmental Pointer to the preceding Mark.

Figure A shows a typical data stack before and after execution of the Begin Block Entry instruction. Figure B shows a typical data stack before and after execution of a Begin Block Exit instruction.

BENT Begin Block Entry format: "53"

The contents of the active stack head registers are pushed into the top of the stack in memory. The Stack Pointer, SP, is then increased by eight to begin the construction of the four-word Mark. The contents of the Environmental Pointer, EP, are copied into SP-6, the first word of the Mark under construction. If the DLINK entry of the current Mark is odd-valued, it is copied into SP-4, the second word of the Mark under construction. If that DLINK value is even, the (contents of EP) +1 is loaded into SP-4.

The Environmental Pointer is then set to SP-6, activating the Begin Mark which has been constructed.

BXIT Begin Block Exit format: "58"

The stack head registers are all marked inactive. The top of the stack is adjusted to the location preceding the current Mark by loading the Stack Pointer register with the contents of the Environmental Pointer register, minus two.

The previous Mark is then reactivated as the latest Mark by loading the Environmental Pointer with SLINK (first word of the Begin Mark being removed).

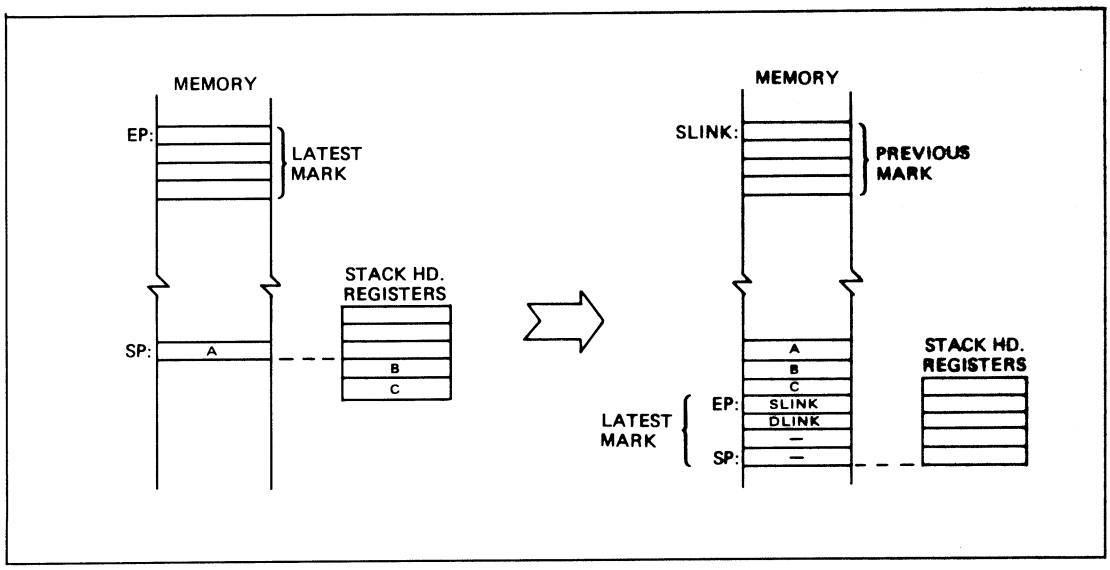


Figure A. Begin Block Entry Instruction.

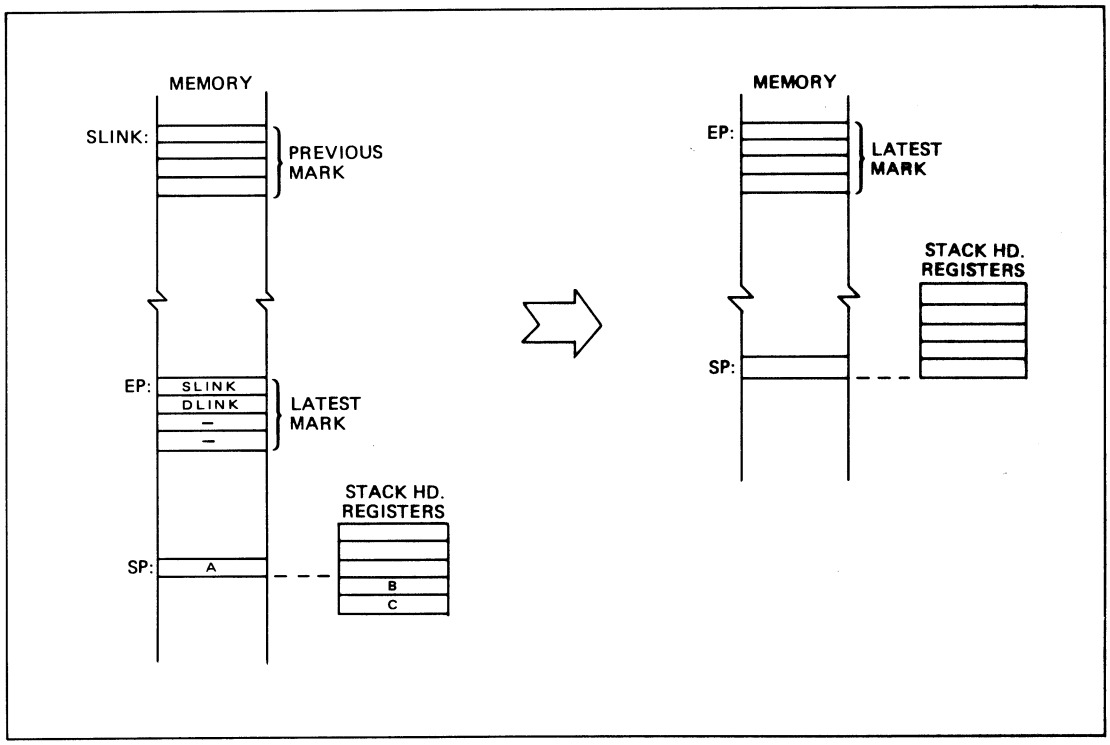


Figure B. Begin Block Exit Instruction.

9.2 MARK STACK FOR PROCEDURE CALL INSTRUCTION

The Mark Stack for Procedure Call instruction initiates the construction of a Procedure Mark in the data stack. (A succeeding CALL instruction completes the construction of the Mark and activates it.)

The Mark Stack for Procedure Call instruction is compiled for an MPL PROCEDURE invocation. Its function is to initiate the construction of a Procedure Mark in the top of the stack. It establishes the SLINK and DLINK entries in the first two words of the Mark under construction, and passes information specifying the location of the called procedure and the number of words to be returned by the called procedure in the last two words of the Mark.

Figure A shows a typical data stack before and after execution of the Mark instruction, and the format of the instruction. The instruction's DESTINATION field and the byte to the left of that field get copied into the Mark under construction.

The rightmost 16 bits of the Mark instruction, DESTINATION, specifies the entry point into the called procedure. If the called procedure is in the currently active program segment, the DESTINATION is a Program Pointer value; if the called procedure is in a remote program segment, the DESTINATION field is a Program Library Number, PLIBN, and Program Reference Table Number, PRTN. The Z bit of the instruction specifies the proper interpretation of the DESTINATION field.

The number of words to be returned by the called procedure is specified in the R field of the instruction. This parameter is left in the Procedure Mark for use by the Exit instruction which removes the Mark.

The value of SLINK, the link to the Mark of the next outermost block for the called procedure, is computed during the instruction execution. The DLEX field of the instruction specifies the number of levels of indirect addressing, up the chain of SLINK entries, to this Mark. (See topic 2.11.)

MARK Mark Stack for Procedure Call format: "50 xx xx xx"

The active stack head registers are pushed into the top of the stack in memory. The Stack Pointer, SP, is then increased by eight to begin the construction of the four-word Mark.

The second (from the left) byte of the instruction (DLEX, Z, R) is loaded into the rightmost byte of the word at SP. The rightmost two bytes of the instruction, DESTINATION, are loaded into SP-2. The contents of the EP register are loaded into the word at SP-4, to become the DLINK.

The SLINK field is then determined by tracing back through the linked list of SLINK entries by DLEX levels, counting the contents of the EP register as the 0 level, the SLINK in the current Mark as the first level, etc. The SLINK entry, which is DLEX levels up the list, becomes the SLINK value loaded into the location at SP-6.

The specific procedure for determining SLINK is as follows:

1. The DLEX field of the instruction is placed in a temporary register, T1. The value in the EP register (i.e., the Stack Base relative address of the first word of the current Mark) is placed in another temporary register, T2.

2. If the value in T1 is zero, then the contents of T2 are loaded into location SP-6, to form the SLINK entry; execution of the Mark instruction is then completed.

3. If the value in T1 is not zero, then the contents of the word pointed to by T2 replace T2; T1 is decremented by one, and processing of the instruction continues with step 2, above.

The specific definitions of the instruction fields are given in Figure A.

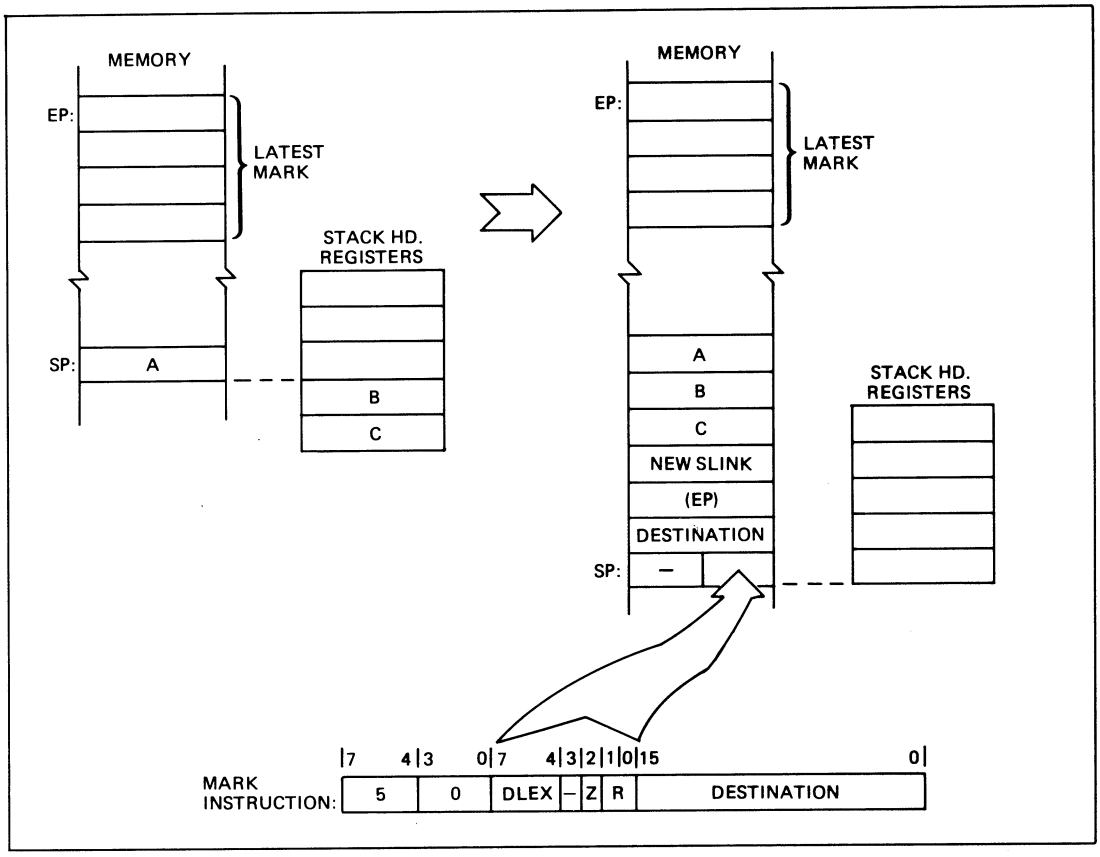


Figure A. Mark Stack for Procedure Call Instruction.

9.3 PROCEDURE CALL INSTRUCTION

The Procedure Call instruction completes the construction of a Procedure Mark in the data stack and activates it. (The construction of the Mark was initiated by a preceding Mark instruction.)

The Procedure Call instruction is compiled for an MPL PROCEDURE invocation. Its function is to complete the construction of a Procedure Mark, begun by a preceding Mark instruction, and to activate the called procedure.

Figure A shows a typical data stack before and after execution of the Procedure Call instruction, along with the format of the instruction. The "before" picture of the stack indicates the Mark whose construction was begun by the preceding Mark instruction, and a sequence of words which were pushed into the stack by code following the Mark instruction. These words are to be passed to the called procedure as arguments.

One of the functions of the Procedure Call instruction is to activate the new Procedure Mark by adjusting EP to point to the base of this Mark. Since the SP has been advanced as arguments were pushed into the stack, the Procedure Call instruction must indicate how many argument words are being passed. This information is provided in the MARK BASE field, the rightmost byte of the instruction. Actually, the MARK BASE is a quantity which is three greater than the number of words passed as arguments, and is therefore the number of words between the base of the new Mark and the top of the stack.

The other function of the Procedure Call instruction is to complete the construction of the new Procedure Mark. The before and after pictures of this Mark are shown in Figure B. The contents of the Mark under construction are defined in topic 9.2, and the contents of the completed Mark are defined in topic 2.8. The Procedure Call instruction extracts the destination information from the "before" Mark, and using the Z bit of this Mark to interpret this information, changes the Program Base and Program Length registers (if a new program segment is being invoked), and loads the new value into the Program Pointer. This accomplishes the actual jump of the program to the called procedure.

The construction of the Mark is completed by the Procedure Call instruction storing the current Program Pointer, Program Library Number, and overflow status into the Mark.

CALL Procedure Call format: "52 xx"

The active stack head registers are pushed into the stack in memory.

EP is loaded with the value SP-2* MARK-BASE. (EP now points to the base of the Mark.)

The DESTINATION word is fetched from location EP+4.

The current PP is stored at location EP+4. This saves the return address.

The word at location EP+6 is fetched and the Z bit (bit 2) is saved. The current PLIBN and overflow status (from the PSR) and the R field (from the word at location EP+6, bits 1-0) are merged to form the status word and loaded into location EP+6. The overflow status in the PSR is reset.

If the Z bit location (EP+6, bit 2) is a zero, then the DESTINATION is stored into PP and the CALL instruction is complete. If the Z bit is a one, the called procedure is in a remote program segment and processing of the instruction continues.

DESTINATION, bits 15-8, becomes the new PLIBN field in the PSR.

The new PLIBN is used as an index into the Program Library and the program segment's descriptor is fetched from PLIB. The PB and PL registers are set from the values in the descriptor.

If the attention bit in the program segment's descriptor is set, an attention interrupt is caused, and execution of the Procedure Call instruction is terminated; otherwise the instruction execution continues.

If the trace bit in the program segment's descriptor is set, the internal trace interrupt status bit is set. (See topic 2.17.)

The value of the remote PRTN (DESTINATION, bits 7-0) is used as an index into the remote procedure's PRT. This word (external entry address) is fetched and placed into PP.

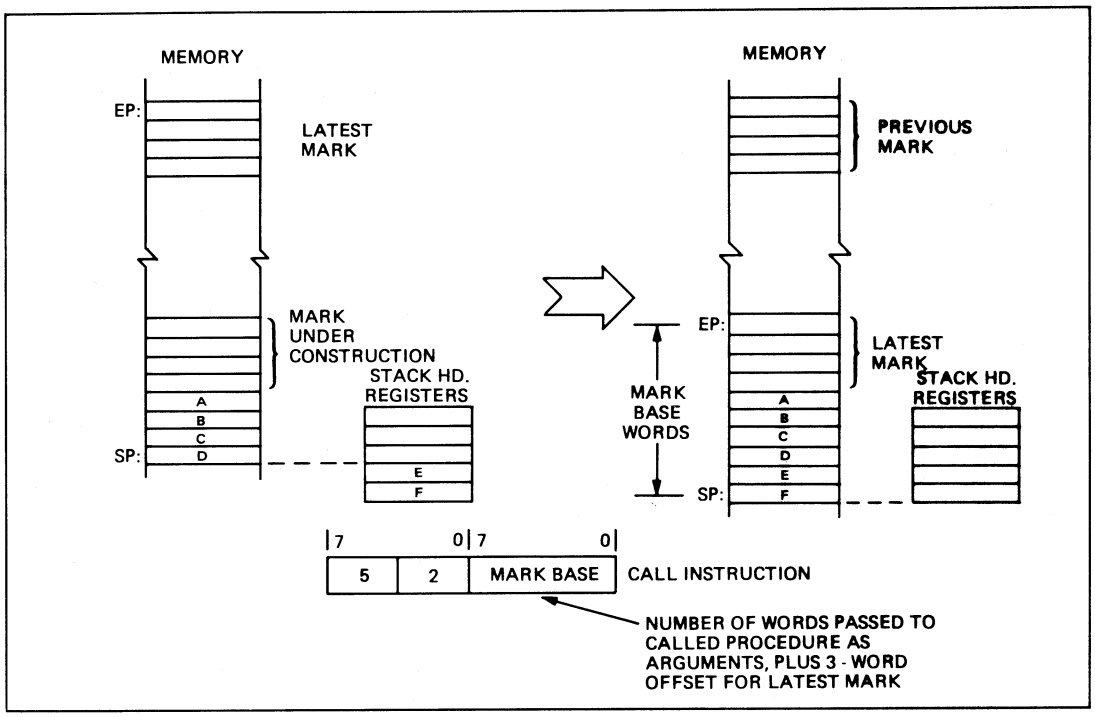


Figure A. Procedure Call Instruction.

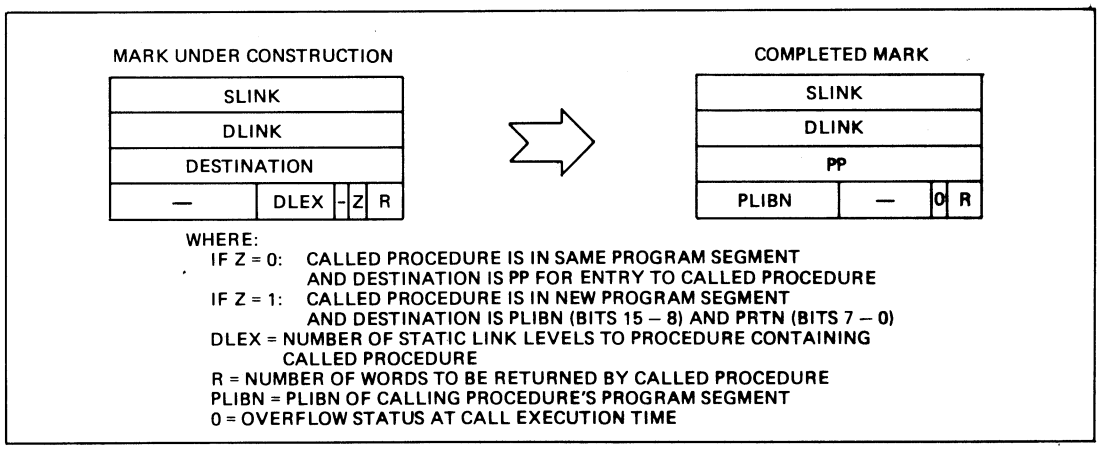


Figure B. Procedure Mark Completion by Call Instruction.

9.4 PROCEDURE BLOCK EXIT INSTRUCTION

The Procedure Block Exit instruction removes the Procedure Mark upon exiting a called procedure.

The Procedure Block Exit instruction is compiled for the MPL END or RETURN statement which terminates a PROCEDURE block. Its function is to "roll back" the stack to the environment which existed when the procedure was called. Since the program may be executing within one or more BEGIN blocks when the PROCEDURE block is to be exited, rolling back the environment may remove one or more Begin Marks along with the Procedure Mark itself.

Figure A shows a typical data stack before and after execution of the Procedure Block Exit instruction. In this example, it is assumed that no BEGIN blocks are being executed at the point when the PROCEDURE block is being exited.

The Procedure Block Exit instruction not only rolls back the stack by deactivating the latest Procedure Mark (and any subsequent Begin Marks), but it also restores the program environment and the Program Status Register. Finally, it leaves the R words returned by the procedure (R is specified in the Procedure Mark) in active stack head registers, so that they will remain on the top of the stack when the Mark is popped.

A flow chart for determining how many Marks are to be removed is shown in Figure B. If the DLINK value (in the Mark being removed) is "FFFF" this indicates that the Mark is that of the MAIN PROCEDURE block (or, if it is a Begin Mark, that the next outermost PROCEDURE block is a MAIN PROCEDURE). Exiting of this procedure means that the program execution has been completed, and therefore a program complete interrupt, interrupt vector number 5 is to be generated.

If the DLINK value is odd, this indicates that the latest Mark is a Begin Mark. The DLINK value of the first Begin Mark following a Procedure Mark is the SLINK value, plus one, of that Begin Mark. All consecutively successive Begin Marks have DLINK entries which are copies of that DLINK. Therefore, the instruction determines the location of the latest Procedure Mark by subtracting one from the DLINK entry of a Begin Mark.

If the DLINK value is even, the latest Mark must be the Procedure Mark which is to be removed.

EXIT Procedure Block Exit format: "54"

The DLINK word at location EP+2 is fetched. If it is "FFFF" a program complete interrupt (interrupt vector number 5) is generated; instruction execution is then terminated. If it is an odd value, one is subtracted from it, and the result is placed into the EP register.

The stack is adjusted so that R words are in the stack head registers (the value of R is contained in bits 1, 0 of the word at location EP+6).

The overflow status in bit 2 of the word at location EP+6 is ORed into the current overflow status bit of the Program Status Register.

The PP word in location EP+4 replaces the contents of PP.

The PLIBN byte in bits 15-7 of location EP+6 is fetched and saved in a temporary register.

The DLINK value in the EP register, minus two, is loaded into the SP register, and the contents of location EP+2 (fetched above) is loaded into the EP register.

If the value of PLIBN fetched above equals the contents of the current PLIBN in the PSR, then the processing of the instruction is complete; otherwise the processing of the instruction continues.

PLIBN is used as an index into the Program Library and the program segment descriptor is fetched. The PB and PL registers are loaded with the values from the Descriptor.

If the attention bit, A, in the descriptor is set, an attention interrupt (interrupt vector number 8) is taken; otherwise, if the trace bit is set, the trace status bit is set.

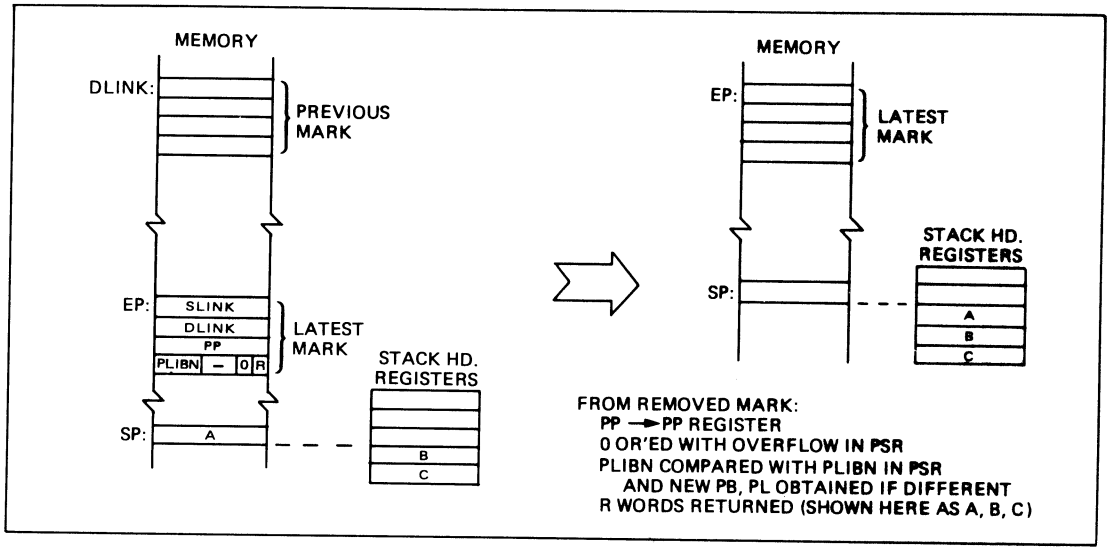


Figure A. Procedure Block Exit Instruction, Where Latest Mark is a Procedure Mark.

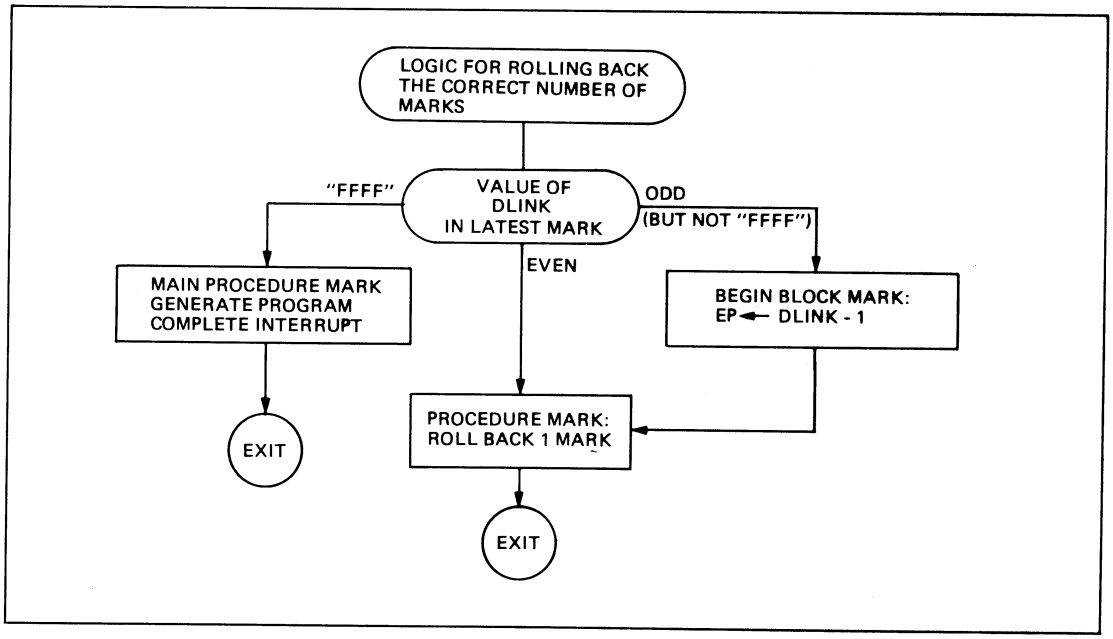


Figure B. Number of Marks Rolled Back by Procedure Block Exit Instruction.

9.5 INTERRUPT PROCEDURE EXIT INSTRUCTION

The Interrupt Procedure Exit instruction removes the Interrupt Mark upon exiting from a procedure entered as a result of an interrupt.

The Interrupt Procedure Exit instruction is compiled for the MPL END and RETURN statements which terminate an INTERRUPT PROCEDURE block. Its function is to "roll back" the stack to the environment which existed when the interrupt was acknowledged. Since the program may be executing within one or more BEGIN blocks when the interrupt procedure is to be exited, rolling back the environment may involve rolling back one or more Begin Marks before removing the Interrupt Mark itself.

Figure A shows a typical data stack before and after execution of an Interrupt Procedure Exit instruction. In this example it is assumed that no BEGIN blocks are being executed at the point when the INTERRUPT PROCEDURE block is being exited.

The Interrupt Procedure Exit instruction not only rolls back the stack by deactivating the latest Interrupt Mark, but it also restores the program environment and the Program Status Register of the previous procedure in the current stack.

An explanation of the process of rolling back Begin Marks, until a Procedure Mark is reached, is given in topic 9.4. The same process is used to roll back to an Interrupt Mark.

IXIT Interrupt Procedure Exit format: "55"

The execution follows these steps:

1. The active stack head registers are pushed into the stack in memory.

2. If the processor is in normal mode, bits 3 and 2 of the byte at location EP + 7 are copied into bits 3 and 2, respectively, of the PSR. This restores the carry and overflow status. The process continues with step 3. If the processor is in the executive mode, the byte at location EP + 7 replaces the lower half of the PSR. This restores the previous interrupt mask, carry, overflow, and mode (executive/normal) status.

3. The SP register is set pointing to EP-2 and the contents of location EP + 2 are loaded into the EP register.

4. A new PLIBN is installed into the Program Status Register from the word at SP, bits 15-8. This is used as an index into PLIB to retrieve PB and PL values. These values are loaded into the PB and PL registers.

5. If the attention bit in PLIB is not set, or if the attention interrupt (interrupt vector number 8) is not armed, the process continues with step 6. If the attention bit is set, and the interrupt is armed, an attention interrupt is then processed, and execution of the Interrupt Procedure Exit instruction is terminated; otherwise the instruction execution continues.

6. If the 'Z' bit in the new status, (location EP + 6, bit 0) is set, the PRTN (from location EP + 4) is used to index into the PRT; the address retrieved from the PRT is loaded into the Program Pointer Register. If the 'Z' bit is not set, the word at location EP + 4 is loaded into PP.

7. If the trace bit in PLIB is not set, the instruction execution is complete. If the trace bit is set, a trace flag is set to cause an interrupt after the next instruction. All interrupts are postponed until after the trace interrupt. The instruction execution is then complete.

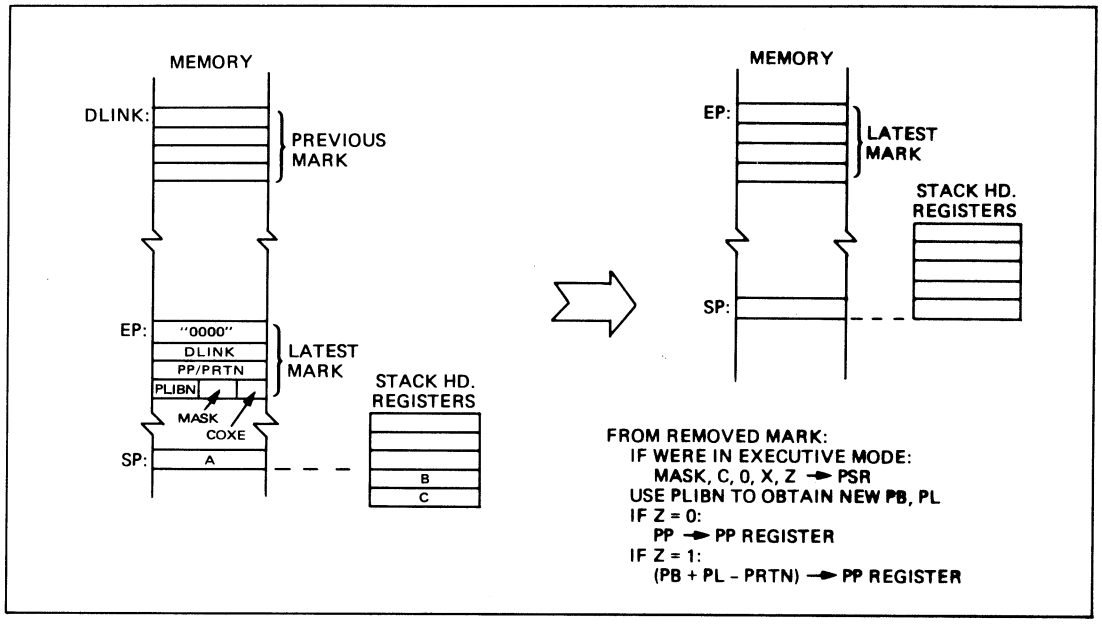


Figure A. Interrupt Procedure Exit Instruction.

9.6 RESUME TASK IN ANOTHER STACK INSTRUCTION

The Resume Task in Another Stack instruction inactivates the current data stack and activates a new data stack. The process involves capping off the current stack with an Interrupt Mark and removing an Interrupt Mark from the top of the new stack.

The function of the Resume Task in Another Stack instruction is to switch to the environment in the top of a presently inactive data stack. A currently inactive stack has previously been capped off by an Interrupt Mark which supplies the Program Status Register, Program Pointer, and Environmental Pointer register values for the new environment. The currently active stack is then capped off with an Interrupt Mark, to retain its corresponding parameters, and its Stack Pointer register value is saved in its Stack Base location. The currently active stack provides the Stack Base and Stack Length register values for the stack which is to be activated.

Figure A shows one typical data stack being inactivated and another typical data stack being activated, by execution of this instruction. Note that the TOS of the currently active stack contains the Stack-Base relative address, SD, of a pair of locations which contain the Stack Base value (divided by four) and the Stack Length value for the stack which is to be activated.

RESM Resume Task in Another Stack format: "56"

The execution follows these steps:

1. If the system is not in the executive mode a privileged instruction violation interrupt (interrupt vector number 9.5) is generated immediately, and instruction execution is terminated.

2. The address of the stack descriptor, SD, is saved in a temporary register, TI, and is popped from the top of the stack.

3. The active stack head registers are pushed into the stack in memory.

4. The value in the Stack Pointer, SP, register is increased by eight, and is stored in the location pointed to by SB.

5. A zero is stored in the word at SP-6. The contents of EP are stored in the word at SP-4. The contents of PP are stored in the word at SP-2. The PSR is stored in the word at SP.

6. The interrupt stack active status bit is reset. The stack descriptor address is fetched from the temporary register T1 and the SB and SL registers are set from the values in the descriptor.

7. The SP register is set to the contents of the word pointed to by SB. The EP register is set to the value of SP-6. EP and SP now point to the first and fourth word (respectively) of an Interrupt Mark.

8. Execution continues with step 2 of the Interrupt Procedure Exit instruction (see topic 9.5).

This is a privileged instruction.

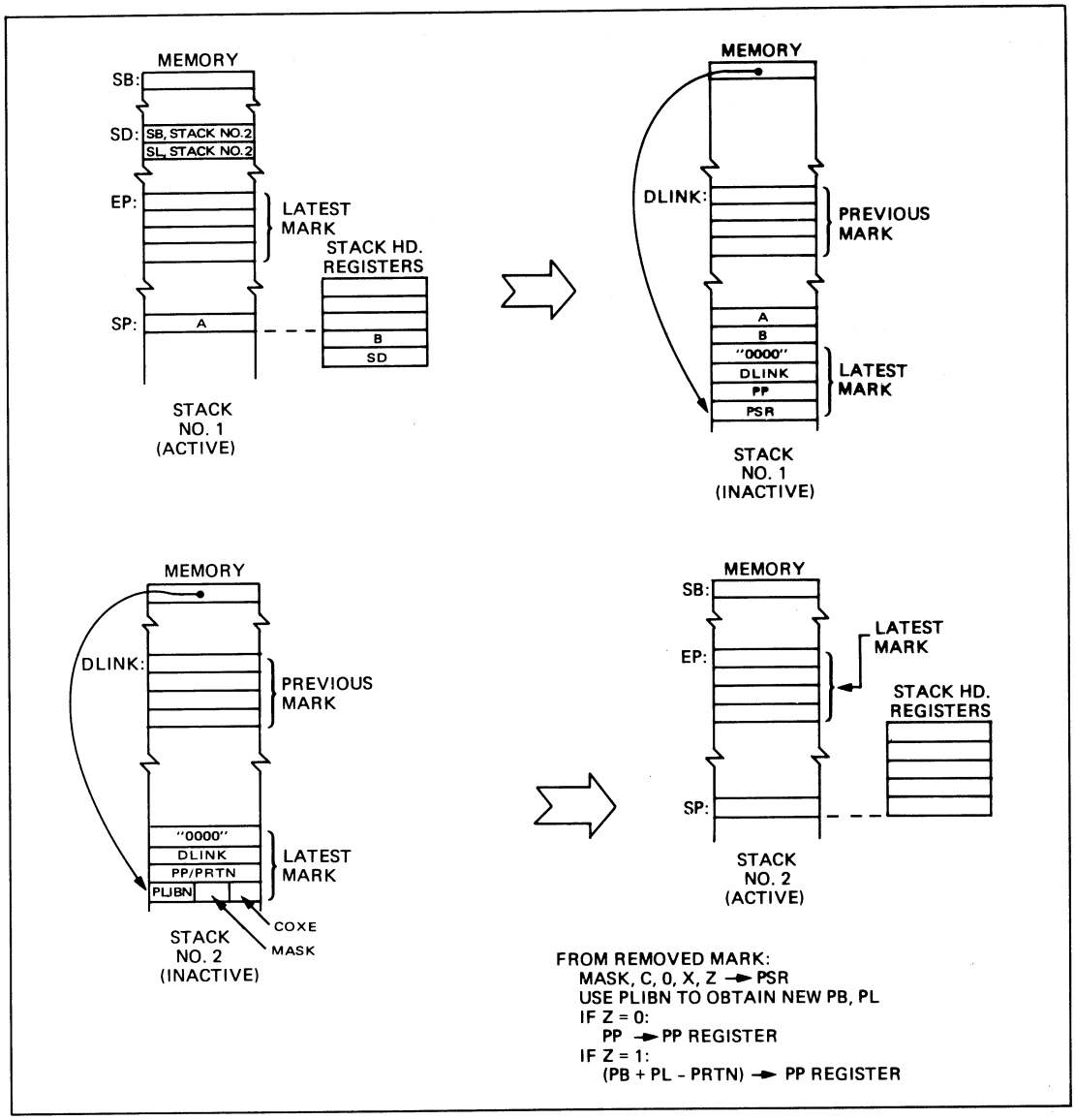


Figure A. Resume Task in Another Stack Instruction.

9.7 WAIT FOR AN INTERRUPT INSTRUCTION

The Wait for an Interrupt instruction generates an Interrupt Mark and places the processor in the wait mode until an interrupt occurs.

The Wait for an Interrupt instruction places an Interrupt Mark on the top of the data stack. It then sets the processor in the wait mode, causing instruction execution to stop. When any interrupt occurs, the wait mode is reset, and instruction execution continues with the processing of an interrupt. See Figure A.

WAIT Wait for an Interrupt format: "5C"

The active stack head registers are pushed into the data stack in memory. The value in the Stack Pointer, SP, is increased by eight. The value in SP is loaded into the location pointed to by the Stack Base register.

The Interrupt Mark is installed. The Program Status register is loaded into the location at SP. The Program Pointer register is loaded into the location at SP-2. The Environmental Pointer register is loaded into the location at SP-4.

The wait mode bit is set and instruction execution ceases.

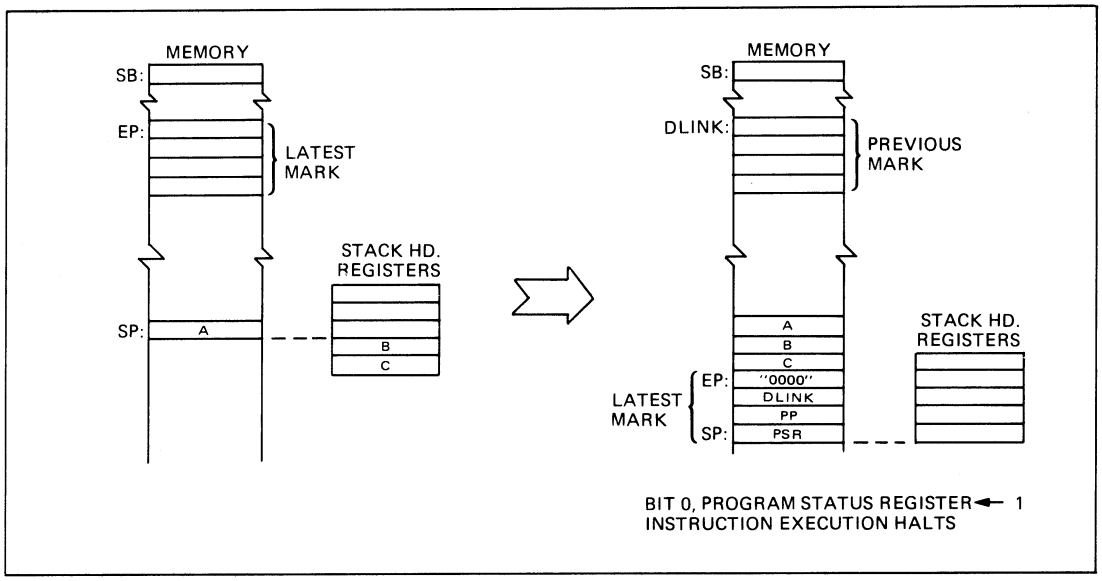


Figure A. Wait for an Interrupt Instruction.

9.8 SUPERVISOR CALL INSTRUCTION

The Supervisor Call instruction generates an Interrupt Mark and a Supervisor Call Interrupt.

Control is passed to the Supervisor system software by executing a Supervisor Call instruction. This process is accomplished by generating a Supervisor Call interrupt, interrupt vector 4.

If the current data stack is the interrupt data stack (the data stack used by the Supervisor Procedure), an Interrupt Mark is installed, but the stack remains active. This is shown in Figure A.

If the current data stack is not the interrupt data stack, and the interrupt is to be taken in the interrupt stack, an Interrupt Mark is installed, and the current stack is made inactive. The interrupt data stack is then activated. This is shown in Figure B.

In either of the above two cases a one word argument, which was in the top of the previously active stack, is pushed into the top of the new active stack.

SUPV Supervisor Call format: "09"

The word in the top of the stack is saved in a temporary register, T1.

The active stack head registers are pushed into the data stack in memory.

The value of the Stack Pointer, SP, is increased by eight.

The value in SP is loaded into the location pointed to by the Stack Base register.

The Interrupt Mark is installed. The Program Status Register is loaded into the location at SP. The Program Pointer register is loaded into the location at SP-2. The Environmental Pointer register is loaded into the location at SP-4.

If the current data stack is not the interrupt data stack, and the interrupt is to be taken in the interrupt stack, the latter stack is activated. The Stack Base and Stack Length for the interrupt data stack are accessed from Monobus locations "00000" and "00002" and loaded into the Stack Base and Stack Length registers. The value in the location pointed to by the Stack Base register is loaded into the Stack Pointer register.

In either case, the argument in T1 is pushed into the currently active stack.

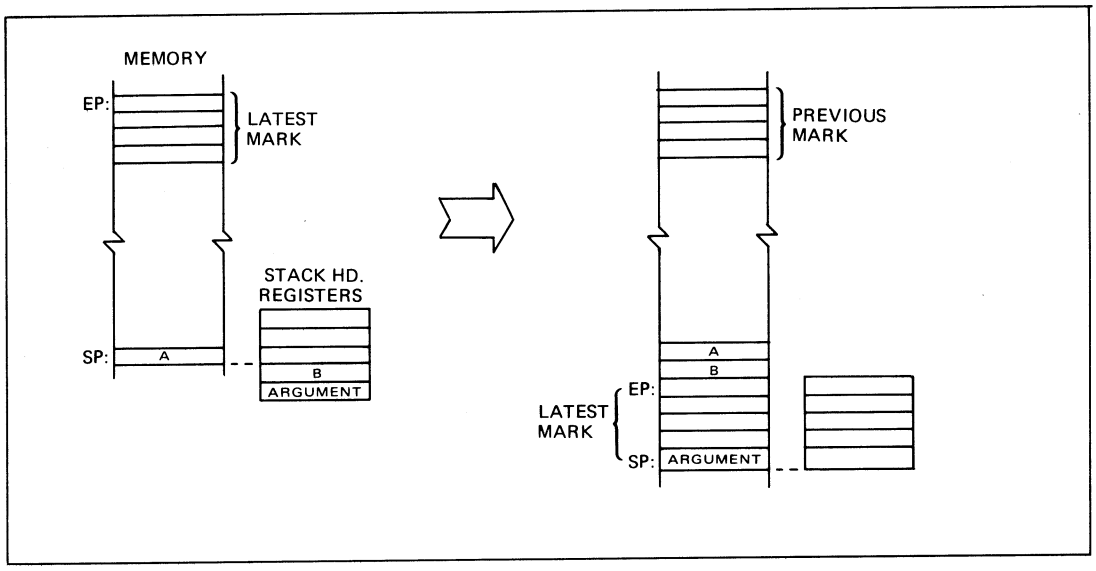


Figure A. Supervisor Call Instruction, User Stack is Interrupt Stack.

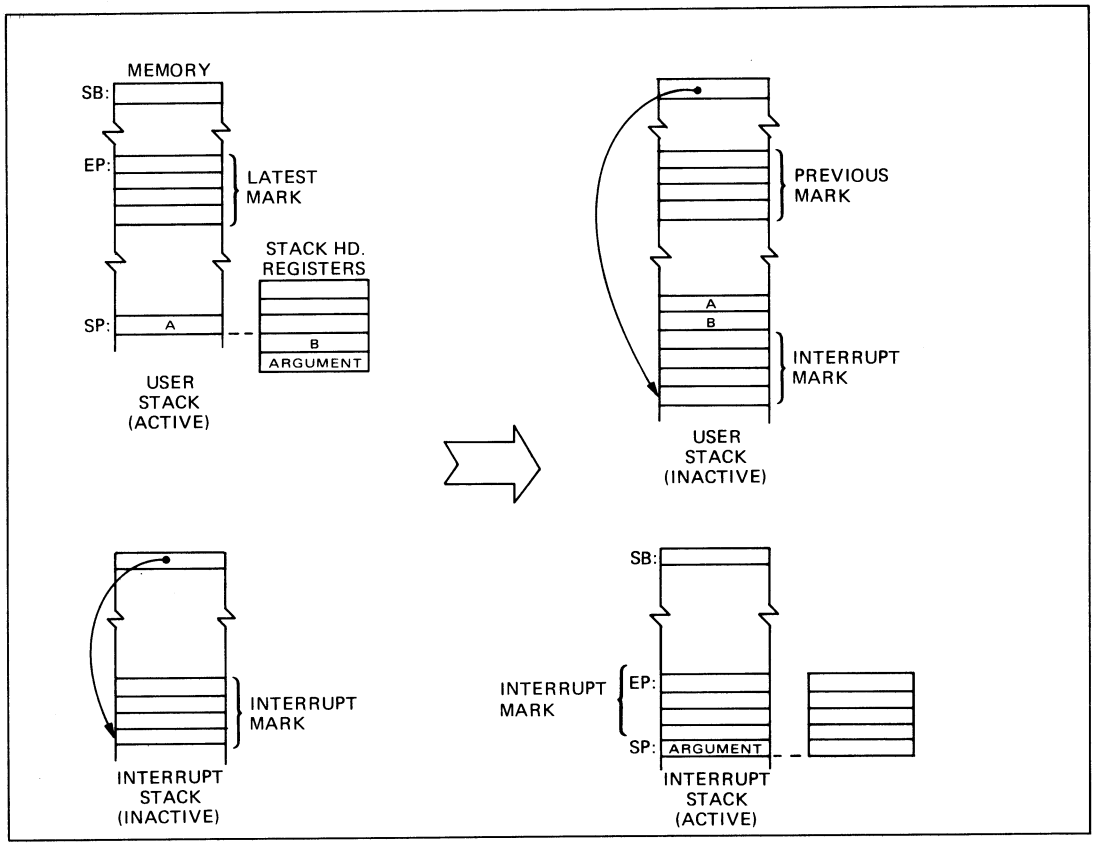


Figure B. Supervisor Call Instruction, User Stack not the Interrupt Stack.

9.9 LOAD ADDRESS INSTRUCTION

The Load Address instruction creates an indirect address to be used in accessing data within the data stack environment of an outer block.

The memory reference instructions are provided with direct addressing modes to access local data within the current environment and to access global data. To access locations within the environment of statically intermediate blocks, these instructions are provided with indirect addressing modes. The indirect address is taken from TOS. (See topic 6.2.) The Load Address instruction is provided to generate this indirect address in TOS. See Figure A.

The indirect address generated by the Load Address instruction is a Stack Base-relative address. The instruction generates this address by adding a 16-bit displacement field, which it provides, to the Stack Base-relative address of the Mark for the desired environment. The Mark's address is determined by tracing back through the SLINK (Static Link) chain from the current Mark by DLEX (Delta Lex) levels. (See topic 2.11.) The instruction specifies the DLEX value to be used.

In addition to generating an indirect address as described above, this instruction can also specify that an index, obtained from TOS, be added in with the indirect address. A field of the instruction specifies whether this index is a byte, word, or doubleword index. The instruction adjusts the index to a byte-level index before adding it into the indirect word.

LADR Load Address format: "06 xx xx xx"

The address displacement field of the instruction, D16, is pushed into the stack.

If the DLEX addition field of the instruction, D, is a 1, the DLEX field of the instruction is used to obtain the SLINK value which points to the Mark for the environment DLEX levels up.

The specific process for determining the desired SLINK value is as follows:

1. The DLEX value is placed in a temporary register, T1. The value in the Environmental Pointer register is placed in a temporary register T2.

2. If the value in T1 is zero, the desired SLINK value is in T2, and the process is completed.

3. The value in T1 is decremented by 1. The value at the location pointed to by the value in T2 is placed in T2.

   The process then continues with step 2, above.

If the Index field of the instruction, I, is a 1, an index in TOS1 (at this point) is to be added to the address being formed in TOS. The Index Length field of the instruction, L, is used to adjust the index. Specifically:

1. If L = 0, the index is ready to be used.

2. If L = 1, the index is multiplied by two.

3. If L = 2, the index is multiplied by four.

4. If L = 3, the index is to be multiplied by six.

The value in TOS is then added into TOS1 and TOS is popped. If the I field is a 0, the procedure described in this paragraph is skipped.

The specific definitions of the instruction fields are given in Figure A.

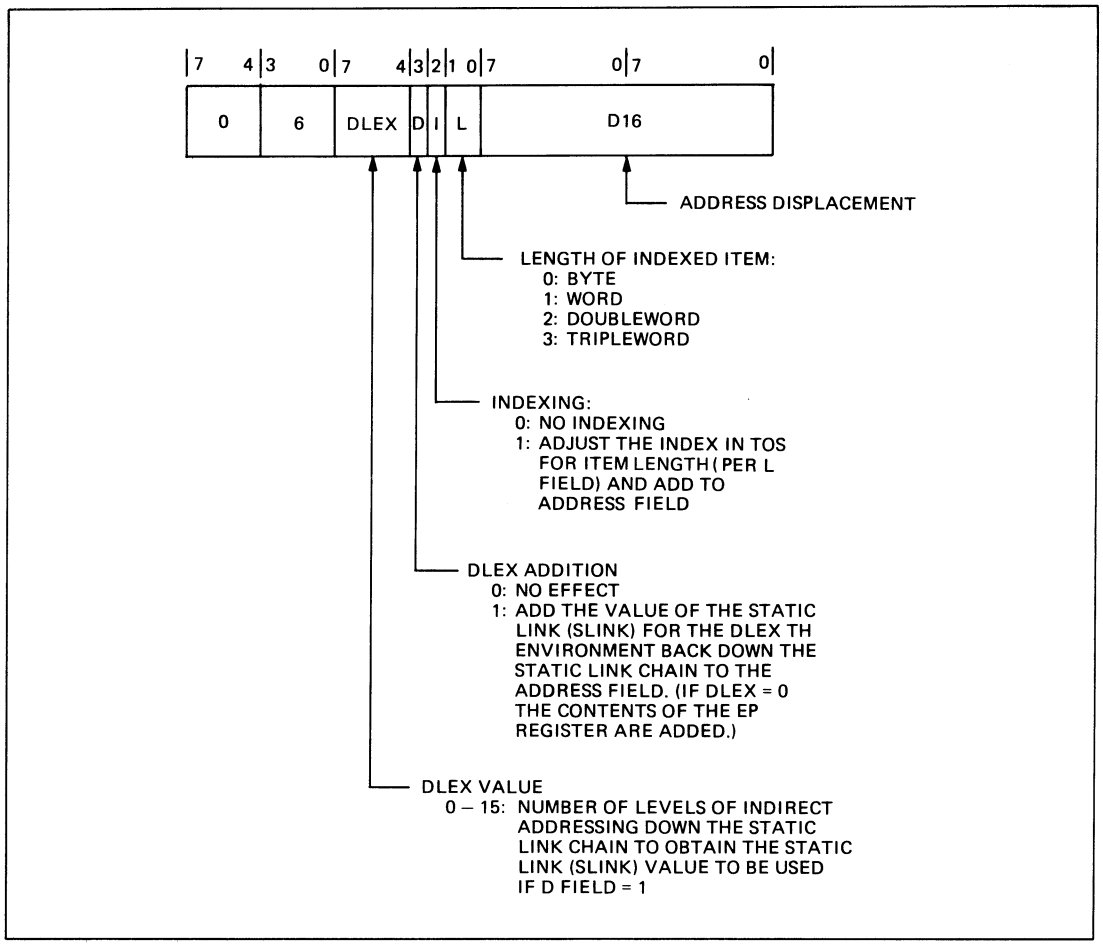


Figure A. Format of Load Address Instruction.

9.10 GO TO INSTRUCTION

The GOTO instruction provides the means to transfer program control and, if desired, to simultaneously exit to the environment of an outer block.

An MPL GOTO statement can specify that program control be transferred to a statement in the current block or an outer block. The GOTO instruction is compiled to implement this process.

The GOTO instruction provides two parameters; the new Program Pointer value and the Delta Lex, DLEX, between the current data stack environment and the desired environment. The desired environment is reestablished by tracing back through the SLINK (Static Link) chain from the current Mark by DLEX levels. The Mark pointed to by that SLINK value is then activated by placing the SLINK value in the Environmental Pointer register.

A typical data stack, before and after execution of the GOTO instruction, is shown in Figure A. Note that the Stack Pointer value is not changed by execution of the GOTO instruction. To complete the environmental "roll back," SP must be set to point to the last word of the new environment. This is done by executing a Set SP instruction at the place that the GOTO instruction points to.

GOTO GOTO format: "57 xx xx xx"

The ADDRESS field of the instruction is placed into the Program Pointer register. The Environmental Pointer, EP, is rolled back DLEX levels. The specific process for doing this is as follows:

1. The DLEX value is placed in a temporary register, T1.

2. If the value in T1 is a zero, the process is complete and instruction execution is completed.

3. The value in T1 is decremented by 1. The value in the location pointed to by the value in EP is placed in EP.

   The process then continues with step 2, above.

The specific definitions of the instruction fields are shown in Figure A.

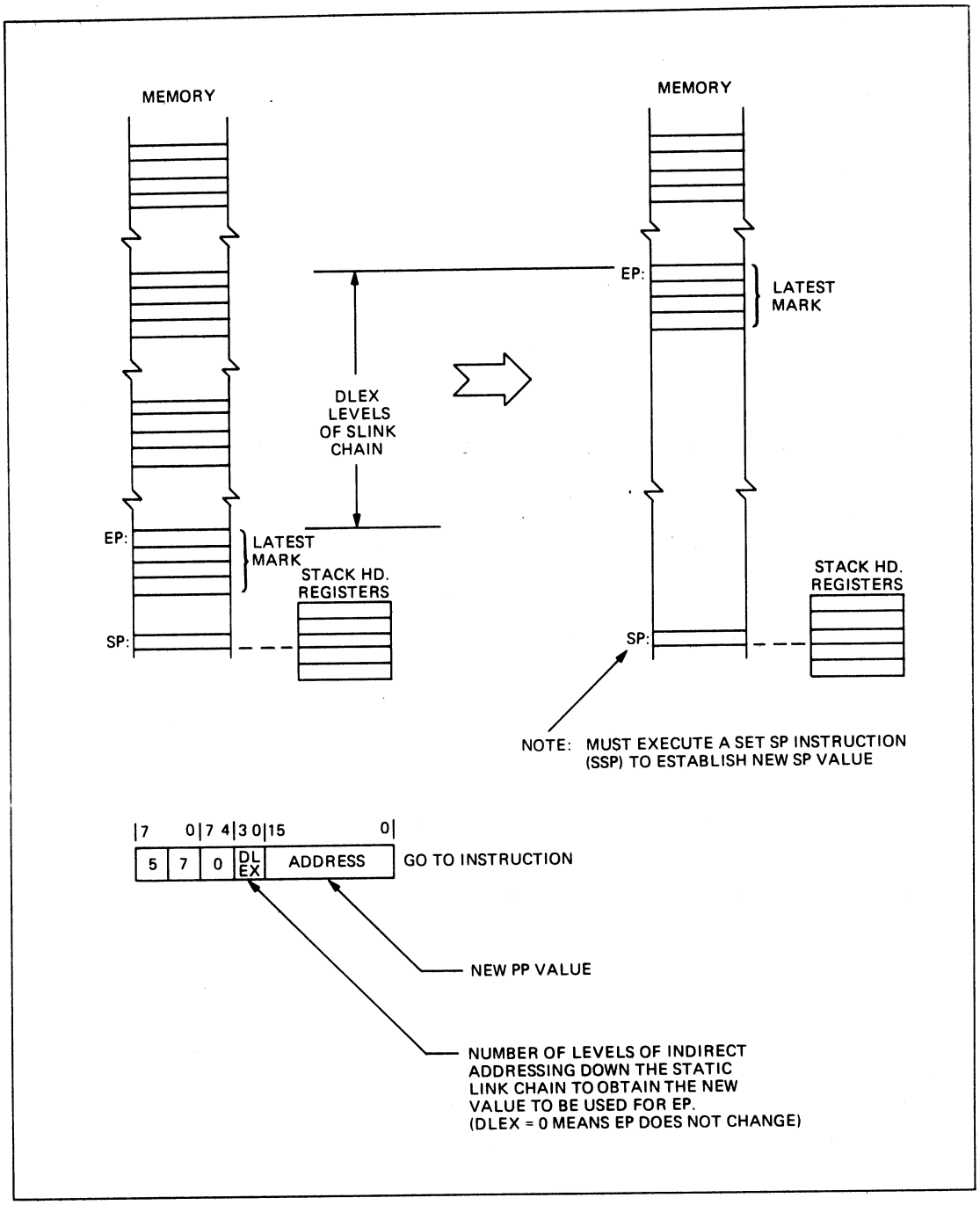


Figure A. Branch to New Environment (GOTO) Instruction.

9.11 MISCELLANEOUS CONTROL INSTRUCTIONS

Ten types of control instructions are defined.

SSP Set Stack Pointer format: "5A xx xx"

The contents of the active stack head registers (if any) are pushed into the data stack within memory. The Stack Pointer register is loaded with the value obtained by adding the contents of the Environmental Pointer to twice the value contained in the rightmost two bytes (xxxx) of the Set Stack Pointer instruction.

SSPI Set Stack Pointer Indirect format: "5B"

The contents of the active stack head registers (if any) are pushed into the data stack within memory. The word on the top of the stack is popped, multiplied by two, and added to the value in the Environmental Pointer; the resultant value is loaded into the Stack Pointer.

SSR Stuff Stack Registers format: "5F"

The contents of the active stack head registers (if any) are pushed into the data stack within memory, (and the Stack Pointer is adjusted accordingly). The stack head registers are then all marked inactive.

POP Pop TOS format: "0B"

The word on the top of the stack is popped.

NOP No Operation format: "01"

This instruction performs no operation. Instruction execution continues immediately with the next sequential instruction.

PNOP Privileged No Operation format: "02"

If the processor is in the executive mode, then this instruction performs no operation. Instruction execution continues immediately with the next sequential instruction.

If the processor is in the normal mode, a privileged instruction violation interrupt, interrupt vector number 9.5 occurs.

This is a privileged instruction.

TCAR Test Carry format: "04"

The carry status condition is tested, and if it is set, a word value of one (True), is pushed onto the top of the stack. If carry is reset, then a zero (False), is pushed. Testing the carry status does not change its condition.

TOVF Test Overflow format: "03"

The overflow status condition is tested, and if it is set, a word value of one (True), is pushed onto the top of the stack. If overflow is reset, then a zero (False), is pushed.

This instruction resets the overflow status condition.

TRAP Trap format: "00"

A Trap instruction interrupt (interrupt vector number 7) is generated immediately after this instruction.

XIM Exchange Interrupt Mask format: "0A"

The Interrupt Mask in the PSR is exchanged with bits 7 through 4 of the word in TOS. The timer update enable bit will be set or reset by bit 3 of TOS. This is the only instruction that can change the state of the timer update enable bit. The instruction following this instruction will be executed before any interrupt can occur.

This is a privileged instruction.

9.12 INITIATE MICROPROGRAMMED PROCEDURE INSTRUCTION

The Initiate Microprogrammed Procedure instruction causes execution of a procedure specified by a firmware routine. This instruction provides the capability to call out procedures incorporated in add-on firmware.

MPL provides the capability to call out procedures which are implemented in firmware added on to the standard 32/S firmware. The capabilities of the standard 32/S can thereby be extended in high-speed firmware and utilized at the MPL level without revising the MPL compiler.

Each 32/S instruction has an operation code which is interpreted to cause a micro-level jump to a specific 32/S firmware routine for executing that instruction. This is diagrammed in Figure A. The Initiate Microprogrammed Procedure instruction provides a 16-bit control memory address which specifies a corresponding jump to a specific firmware routine.

MICR Initiate Microprogrammed Procedure format: "OC xx xx"

The control memory address contained in the rightmost two bytes of the instruction, (xxxx), are loaded into the control memory address register. Execution of the firmware beginning at this control memory address is then initiated.

This is a privileged instruction.

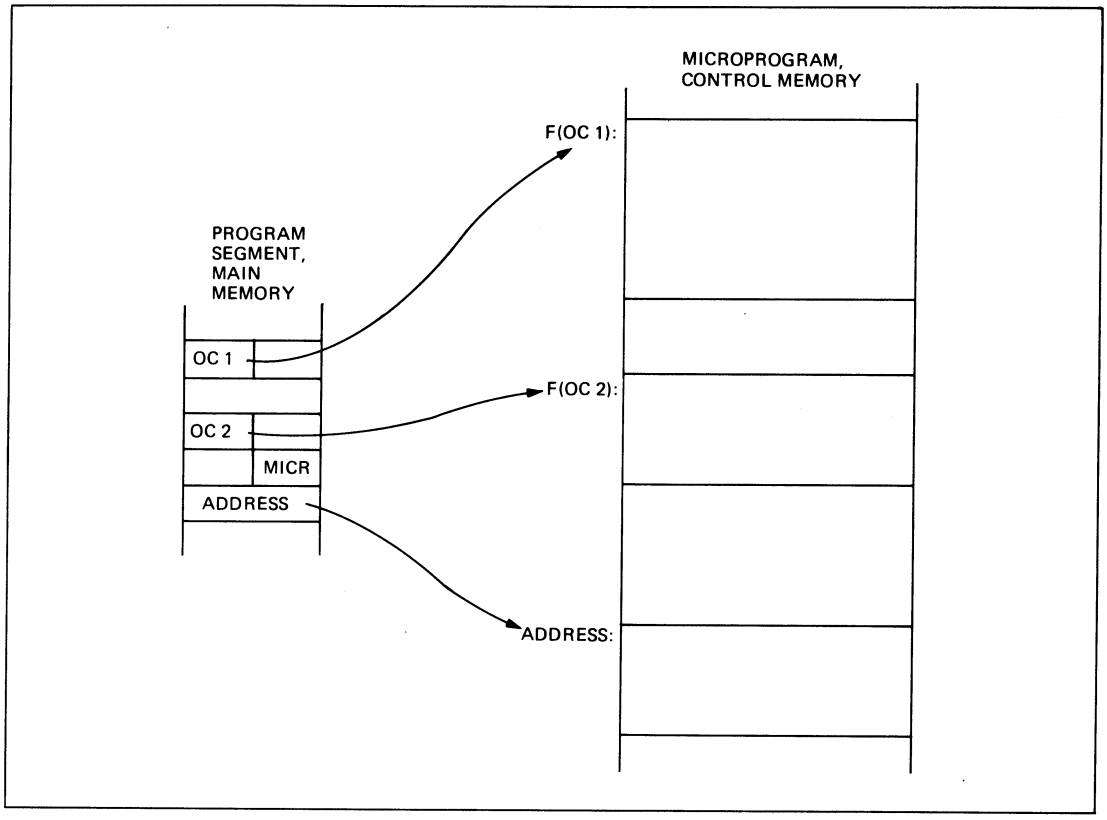


Figure A. Initiate Microprogrammed Procedure Instruction.

10 String Instructions

10.1 STRING DESCRIPTORS AND INSTRUCTIONS

The string move and compare instructions are introduced, and the string descriptor is defined.

The 32/S provides two groups of string instructions: move and compare. The pairs of string operands for these instructions are each defined by a two-word string descriptor.

The format of the string descriptor is shown in Figure A. The most significant 16 bits of the doubleword contain the string start address. Successive bytes of the string are at successively higher addresses. The least significant 16 bits contain the length, in bytes, of the string (0 to 65,535).

The string descriptors for the two strings must be in the top of the stack before execution of the instructions. (In the MVA instruction, two pointers must also be in the top of the stack.) The string descriptors are popped from the stack and used to control the instruction execution. Specifically, the start address is incremented and the string length is decremented as each byte of the strings is moved or compared.

String move instructions move a source string, as defined by the source string descriptor, into the string locations specified by the destination string descriptor. The move is completed whenever either string length is decremented to zero.

String compare instructions scan two strings from left to right until the end of one or both strings, or until a difference is found. Comparisons are made on 8-bit bytes as positive integers. If the strings are equal byte by byte up to the end of one string and the other string is longer, the short string is considered less than the longer string. Strings are equal only if they have identical lengths and each character equals its corresponding character in the other string.

Since the byte-length of the strings being moved or being compared may be quite long, it is necessary to break into the instruction execution in order to service interrupts and concurrent I/O. This is accomplished by stopping the move or comparison operation after 16 bytes have been moved or compared. The Program Pointer (PP) is decremented by two, and the updated string descriptors (and in the case of MVA, the two pointers) are pushed into the stack. The instruction execution is then complete. After any outstanding interrupts or concurrent I/O requests have been handled, execution of the same string instruction continues where it had left off.

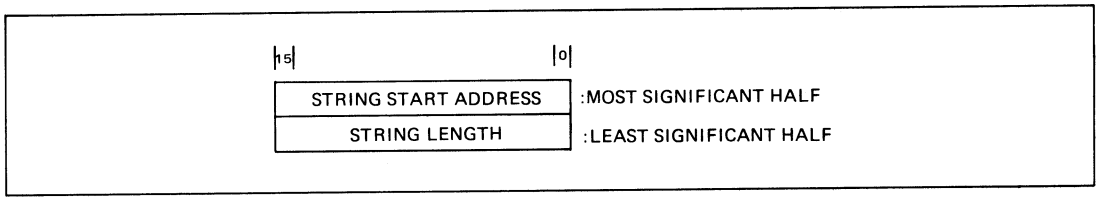


Figure A. String Descriptor.

10.2 STRING INSTRUCTIONS

Three types of move string instructions and six types of compare string instructions are defined.

String Move Instructions

MOV Move String Within Stack Op Code: "4F30"

The string start addresses within both the source string descriptor and the destination string descriptor are relative to the Stack Base, SB.

The source string descriptor (in TOS, TOS1) and the destination string descriptor (in TOS2, TOS3) are popped and used to move a string.

1. If either string length field is zero, the instruction proceeds with step 4.

2. The byte within the stack, addressed by the source start address, is fetched and stored at the byte in the stack addressed by the destination start address.

3. Each address is incremented by one and each length is decremented by one. The instruction proceeds with step 1, unless the move loop (steps 1-3) has been executed 16 times. If the loop has been executed 16 times, the instruction continues with step 5.

4. The updated destination descriptor is pushed into the top of the stack. The instruction execution is complete.

5. The Program Pointer is decremented by two, and the current destination string descriptor and source string descriptor are pushed into the stack. The instruction execution is complete.

MVP Move String from Procedure to Stack Op Code: "4F31"

This instruction is similar to MOV except that the source string is contained in the current Program Segment. (The string start address in the source string descriptor is relative to the Program Base, PB.)

MVA Move String Absolute Op Code: "4F38"

A source string descriptor (in TOS, TOS1), a pointer (in TOS2) to the source string base address, a destination string descriptor (in TOS3, TOS4), and a pointer (in TOS5) to the destination string base address are popped and used to control the move. The pointers are used to fetch the source and destination base address (D18's) from the stack. The start addresses in the string descriptors are offset relative to their respective base addresses. The move occurs similarly to MOV except that if after 16 bytes the instruction is not complete, the pointers as well as the updated string descriptors are pushed back into the top of the stack.

This is a privileged instruction and if executed in normal mode an interrupt (9.5) will be generated instead.

String Comparison Instructions

All string comparison instructions perform similar operations:

1. The two string descriptors in the top of the stack are popped.

2. The string specified by the descriptor which had been in TOS2, TOS3 is compared to the string specified by the descriptor which had been in TOS, TOS1, on the basis specified by the instruction name.

3. If the result of the comparison is true, a word with the value of 1 is pushed into the stack. If the result of the comparison is false, a word with the value of 0 is pushed into the stack.

The string start addresses in both string descriptors are relative to the Stack Base, SB.

In the instruction definitions which follow, only the basis for a true comparison result is given:

SLT String Compare Less Than — Op Code: "4F32"

String of descriptor in TOS2, TOS3 less than string of descriptor in TOS, TOS1.

SLE String Compare Less Than or Equal — Op Code: "4F33"

String of descriptor in TOS2, TOS3 less than or equal to string of descriptor in TOS, TOS1.

SEQ String Compare Equal — Op Code: "4F34"

String of descriptor in TOS, TOS3 equal to string of descriptor in TOS, TOS1.

SNE String Compare Not Equal — Op Code: "4F35"

String of descriptor in TOS2, TOS3 not equal to string of descriptor in TOS, TOS1.

SGE String Compare Greater Than or Equal — Op Code: "4F36"

String of descriptor in TOS2, TOS3 greater than or equal to string of descriptor in TOS, TOS1.

SGT String Compare Greater Than — Op Code: "4F37"

String of descriptor in TOS2, TOS3 greater than string of descriptor in TOS, TOS1.

11 Control Memory

11.1 CONTROL MEMORY

Add-on control memory is addressable via the Monobus, permitting the loading and verifying of firmware in optional writable control memory.

The standard 32/S firmware is provided in 2K (32-bit) words of read-only control memory on the Processor Control board. Additional control memory is available in both read-only and writable control memory modules. New firmware is usually debugged in the writable control memory and then implemented in the lower-cost, non-volatile, read-only memory.

New firmware may be developed, either to add to the 32/S instruction set, or to replace the 32/S instruction set. In either case, the standard 32/S firmware provides the computer control to load and to verify the new firmware in the writable control memory. In the case of a replacement instruction set, control is manually switched from the 32/S firmware to the new firmware (in the writable control memory) after the load and verify operations.

The writable control memory, WCM, interfaces with the computer in two ways. The WCM interfaces to the computer via the Monobus, permitting it to be loaded and verified. The WCM interfaces to the computer via the control memory address and data buses, and operates as control memory.

A manual switch on the Processor Control board is provided to switch out the read-only memory containing the 32/S firmware if a replacement firmware set is being developed.

When interfaced to the computer via the Monobus, the WCM modules respond to Monobus addresses in the "38000" to "3BFFF" range. Control memory address and Monobus address are related as follows (see Figure A):

Monobus address = "38000" + 4 * control-memory-address + byte-number

where

byte-number: byte position within control memory word, with the leftmost byte having a byte position of 0.

The optional read-only control memory module provides up to eight pages of control memory. Its Monobus addressing (for read only) is related to its control memory addressing in the same manner as for the WCM.

NOTE: The WCM modules require a high power supply current. Therefore, if more than one module is used, an auxiliary power supply will be required, the power line etch in the backplane must be cut, and caution must be observed in placement of the modules. See topic 13.1.

Each WCM module provides one page (512 32-bit words) of control memory. A four-pole switch on the module is manually set to specify the page address. The poles of this switch, reading from left to right, are:

S3: bit 2

S2: bit 1

S1: bit 0

S0: not used.

If add-on (to the 32/S) firmware is being developed, the WCM modules must be set to pages 4 through 7 so as to follow the four pages of the 32/S firmware. If a replacement firmware is being developed, the WCM pages may start with page 0.

The procedure followed in debugging 32/S add-on firmware is as follows:

1. Set the page addresses on each WCM module to page 4, or higher.
2. Set the control switch on the Processor Control module to control.
3. Start the computer and load and verify the WCM with the new firmware.
4. Start the computer and debug the new firmware.

The procedure followed in debugging a replacement firmware is as follows:

1. Set the page addresses on each WCM module to page 4, or higher.
2. Set the control switch on the Processor Control module to control.
3. Start the computer and load and verify the WCM with the new firmware.
4. Halt the computer and set the page address on successive WCM to page 0, 1, 2, 3.
5. Set the control switch on the Processor Control module to off.
6. Start the computer and debug the firmware.

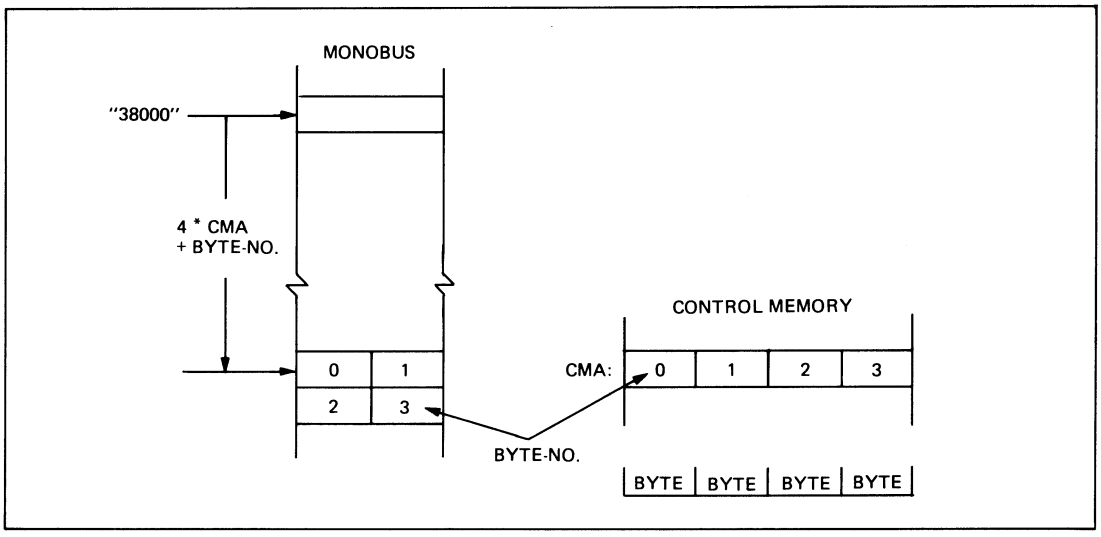


Figure A. Control Memory Address–Monobus Address Relationship.

12 Front Panel

12.1 MAINTENANCE & BASIC FRONT PANELS

The maintenance and basic front panels provide a multi-position key lock switch, and interrupt and load buttons. The maintenance panel, in addition, permits selection of one of 18 different address/data pairs for display or for entry via switches. The maintenance panel also provides 11 control switches and seven status displays.

A simplified view of the maintenance front panel is shown in Figure A. The basic front panel has only the key lock switch, load button, and interrupt button.

The maintenance panel switches and displays are grouped into four areas. The display selectors are 18 switches which define the meaning of the address display, data display, and entry keys in the area immediately below. Each display selector is a momentary switch with an accompanying indicator to mark the last display selector depressed. The definitions of the display selectors are given in topic 12.3. The selectors are labeled with the address definition above, and the data definition below, a horizontal line.

The address displays are 18 indicators which are defined by that display selector which is marked by the lighted indicator. The data displays are 16 indicators which are defined by the marked display selector. The entry keys are 18 alternate action switches which are used to enter the address or the display specified by the marked display selector. Address enter and data enter switches in the control switch area define the use of the entry keys.

The status indicators are seven indicators which display the hardware operational status (e.g., clock running, breakpoint reached).

The control switches are 11 switches which permit the operator to control the computer operation (e.g., system reset) and the panel operation (e.g., address enter).

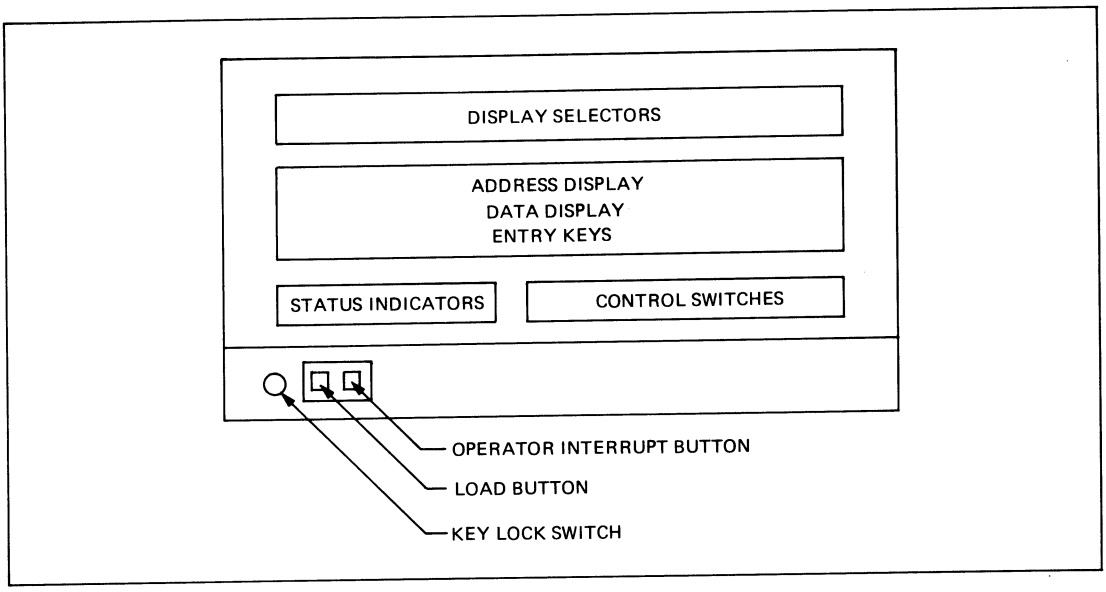


Figure A. Simplified View of Maintenance Panel.

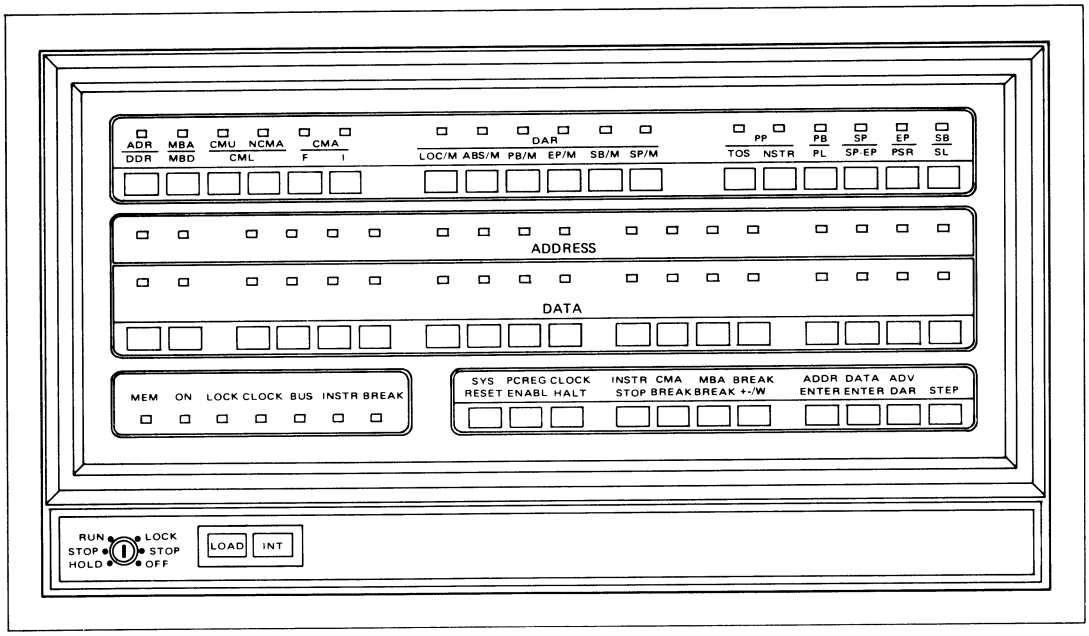


Figure B. Maintenance Panel

12.2 KEY LOCK, LOAD & INTERRUPT BUTTONS

The 32/S computer can be put in any one of five states with the key lock switch. An initial program load can be accomplished by depressing the load button, and an operator interrupt can be caused by pushing the interrupt button.

The key lock switch, load button, and interrupt button are controls which are common to both the maintenance and basic front panels. The meaning of these controls is described in Figure A.

The key lock switch has five unique positions. Basically, OFF and HOLD are identical, except that in the HOLD position, power is supplied to the MOS memory modules (only), permitting data retention without batteries at a time when the user wishes to turn off the computer and controllers. STOP permits all computer operations except instruction execution (e.g., concurrent I/O). RUN and LOCK are identical normal running modes, except that in LOCK the operator interrupt is the only operational button, or switch, on the panel which affects machine execution. While in LOCK any register can be displayed, but only the DAR can be modified (via entry switches or via the ADV DAR switch), and break conditions can be detected (but machine execution does not halt on break).

The specific definitions of the key lock switch positions are as follows:

OFF: All AC and battery power is off.

HOLD: Power supply provides power and refresh signals to the MOS memories, ensuring data retention. (AC line voltage or a battery option must be present.)

STOP: Power is on to the entire computer. All computer operations are enabled except for execution of 32/S instructions. The enabled operations include concurrent I/O transfers, timer update, and power fail interrupt, acknowledgment and operation of the front panel itself. This is the only state in which the LOAD button is active. The purpose of the STOP position is to permit operation of the front panel or to perform an Initial Program Load before instruction execution is begun. It is for this reason that the STOP position is placed both between OFF and RUN/LOCK and between HOLD and RUN/LOCK.

If a power fail occurs while the key is in the STOP position, the computer reverts to the run state. Turning the key from STOP to HOLD returns the machine to the run state and causes a power fail interrupt.

RUN: Power is on to the entire computer. All computer operations, including instruction execution, are enabled. All maintenance panel functions except the LOAD button are enabled.

LOCK: Power is on to the entire computer. All computer operations, including instruction execution, are enabled. All maintenance panel functions which affect machine operation, except the operation interrupt INT are disabled.

The LOAD and INT buttons are defined as follows:

LOAD: LOAD is a momentary-action button which, when depressed, causes an Initial Program Load (IPL) to start. (See topic 12.6.) It is only operative when the key lock is in the STOP position.

INT: INT is a momentary-action button which, when depressed, causes an operator interrupt to occur. (See topic 3.3.) It is operative when the key lock is in the STOP, RUN, or LOCK positions.

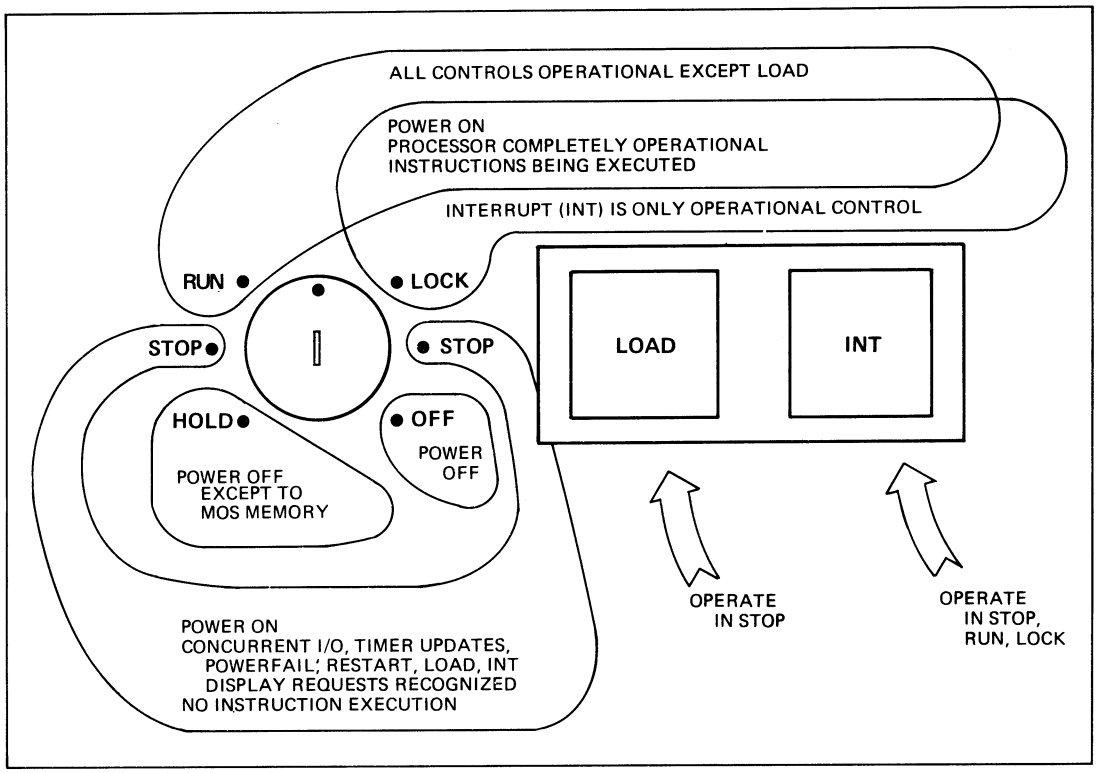


Figure A. Controls Common to Both Basic and Maintenance Panels.

12.3 DISPLAY SELECTORS

The display selectors permit the operator to display and modify Monobus locations specified both absolutely and relative to the 32/S data stack and program segment registers. Local memory and the Program Status Register can also be displayed and modified.

The 18 display selectors are divided into two groups. The leftmost six selectors permit display of hardware-level buses. The rightmost 12 selectors permit display and modification of Monobus locations specified both in absolute addresses and in addresses relative to the data stack and the 32/S hardware registers. These Monobus locations include locations in main memory, read/write control memory, and the Device Register Blocks of I/O device controllers.

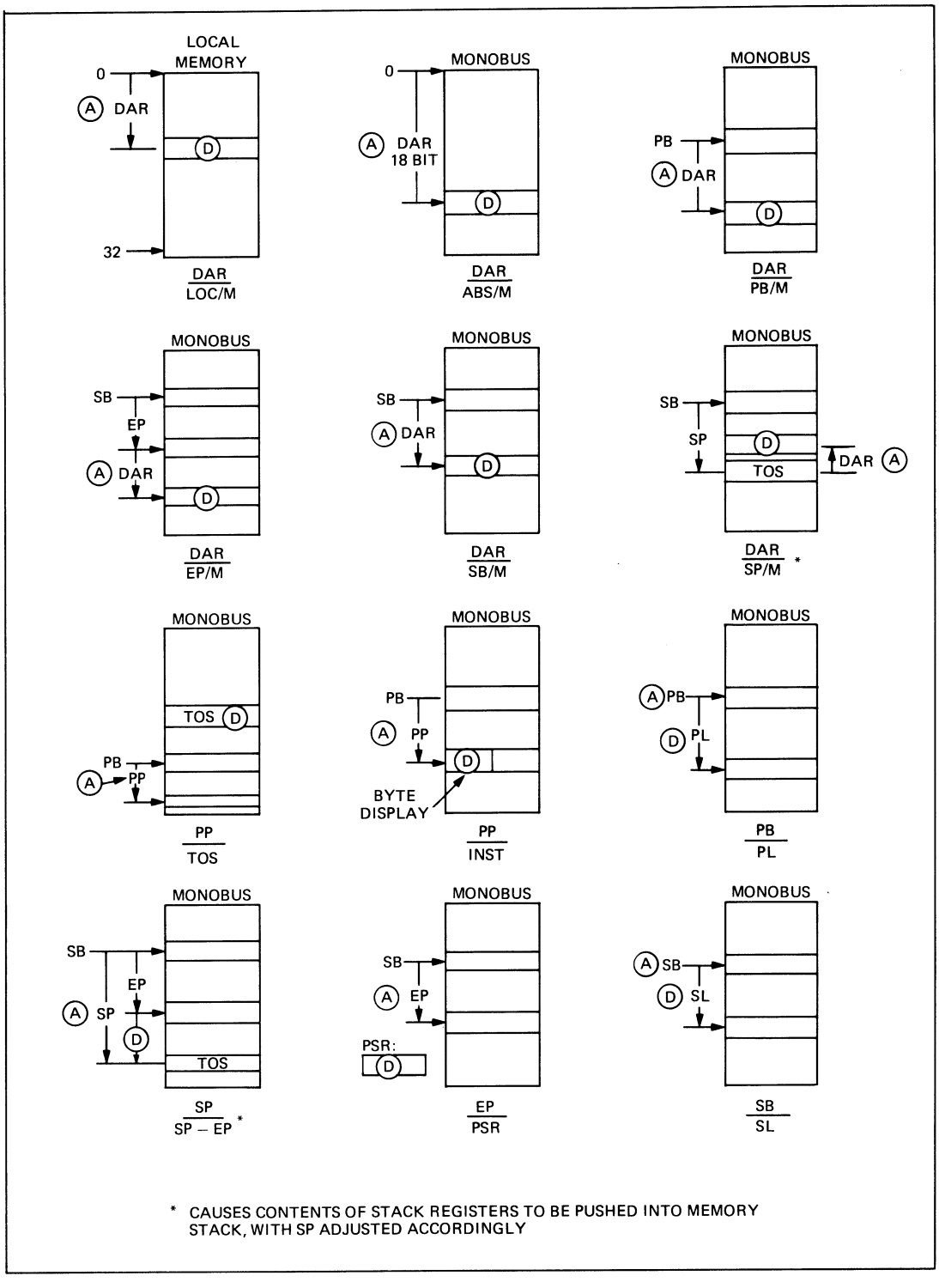
The definition of the rightmost 12 display selectors is indicated in Figure A. A Monobus map is shown with the label from each of the selector switches (except, in the case of the local memory selector, a local memory map is shown). The item which has the (A) symbol next to it is the one which is displayed on the address display and loaded by the ADDR ENTER switch. The item which has the (D) symbol next to, or in it, is the one which is displayed on the data display and loaded by the DATA ENTER switch.

The term DAR refers to the display address register. This is an 18-bit register which is loaded and modified only from the maintenance panel and has no function other than holding a relative Monobus address for panel display and entry purposes. When the DAR is selected, its value can be incremented by depressing the ADV DAR switch, permitting the panel operator to step through Monobus locations.

NOTE: When either of the two display selectors involving the Stack Pointer, SP, are depressed, the contents of the stack head registers are first pushed into the memory stack and SP is adjusted accordingly.

The definition of the six hardware level display selectors is given briefly, although the 32/S user will not need to use them.

CMA / I	:	Display the current control memory address as the address, and the I-bus as the data.
CMA / F	:	Display the current control memory address as the address, and the F-bus as the data.
NCMA / CML	:	Display the next control memory address as the address, and the least significant 16 bits of the control memory data as the data.
CMU / CML	:	Display the most significant 16 bits of the control memory data as address, and the least significant 16 bits of control memory as data.
MBA / MBD	:	Display the hardware Monobus address register as address and the Monobus data register as data.
ADR / DDR	:	Display the most recent contents of the address display and data display registers. (Updated by MBA/MBD and by firmware driven displays.)



* CAUSES CONTENTS OF STACK REGISTERS TO BE PUSHED INTO MEMORY STACK, WITH SP ADJUSTED ACCORDINGLY

Figure A. Dispaly Selection Right-most 12.

12.4 CONTROL SWITCHES

The functions of reset, break-point and single-step are provided in the control switch area. This area also contains the two switches which cause the entry keys to be read in as either address or data.

The 32/S break-point and single-step operations are accomplished with the use of the CLOCK HALT, INSTR STOP, CMA BREAK, MBA BREAK, BREAK+-/W, and STEP switches. The basic definition of these switches' functions is given first, but since their operation interacts, Figure A will be used to explain how break-point and single-step operations are performed.

CLOCK HALT : Depressing this alternate action switch halts the computer clock, stopping execution of firmware instructions.

INSTR STOP : Depressing this alternate action switch generates a stop interrupt and stops further execution of 32/S instructions. Concurrent I/O, timer updates, and the front panel remain operational.

CMA BREAK : Depressing this alternate action switch in conjunction with the MBA BREAK switch specifies that the break address be interpreted as a word address. (By itself, this switch is used to control a break on control memory address - see the manual on the 3200 Microprocessor.)

MBA BREAK : Depressing this alternate action switch specifies that the entry keys provide the Monobus address which defines a break condition.

BREAK+-/W : Depressing this alternate action switch in conjunction with the MBA BREAK switch specifies that the break condition only arises when a write operation is being performed at the Monobus break address. (This switch is also used to control a control memory break function - see the manual on the 3200 Microprocessor.)

STEP : Depressing this momentary action switch causes execution of one firmware or one 32/S instruction depending, respectively, upon whether the CLOCK HALT is depressed or if only the INSTR STOP switch is depressed.

Figure A shows how the six control switches defined above are used to accomplish break-point and single-step operations. The CMA BREAK, MBA BREAK, and BREAK+-/W switches define the break condition. If no break condition is specified, the STEP switch can be used to single step through firmware instructions (if CLOCK HALT is depressed) or 32/S instructions (if INSTR STOP, only, is depressed).

If a break condition is specified, but neither CLOCK HALT or INSTR stop is depressed, the computer will continue to run. However, a sync pulse is generated and the BREAK display (in the status indicators area) flashes each time the break condition occurs.

If a break condition is specified, and either the CLOCK HALT or INSTR STOP is depressed, the computer runs until the break condition occurs. At that point, it gives a stop interrupt (if the INSTR STOP switch, only, is depressed) or halts the clock (if the CLOCK HALT switch is depressed). Once the break condition has been reached, depressing the STEP switch releases the computer to do at least one 32/S instruction (if the INSTR STOP switch, only, is depressed) or at least one firmware instruction (if the CLOCK HALT switch is depressed).

The inputting of address or data with the entry keys is controlled by the ADDR ENTER and DATA ENTER switches, in conjunction with the display selectors. The display address register, DAR, which holds a relative Monobus address for display, is advanced by the ADV DAR switch.

ADDR ENTER : If this momentary action switch is depressed, the 18 bits specified by the entry keys are entered into the address register specified by the active display selector. A depressed entry key is entered as a 1-bit.

DATA ENTER : If this momentary action switch is depressed, the 16 bits specified by the 16 rightmost entry keys are entered into the local memory location, Monobus location, or register specified by the active display selector. A depressed entry key is entered as a 1-bit.

ADV DAR : If this momentary switch is depressed, when any display selector DAR/xxx is selected, the maintenance panel's DAR register is incremented by two, unless LOC/M is selected in which case it is incremented by one.

The system reset function is performed by the SYS RESET switch:

SYS RESET : Depressing this momentary switch issues a system reset signal to all logic in the computer, maintenance panel, and I/O device controllers.

The PCREG ENABL switch is used in conjunction with firmware debug operations only - see the Microprocessor manual.

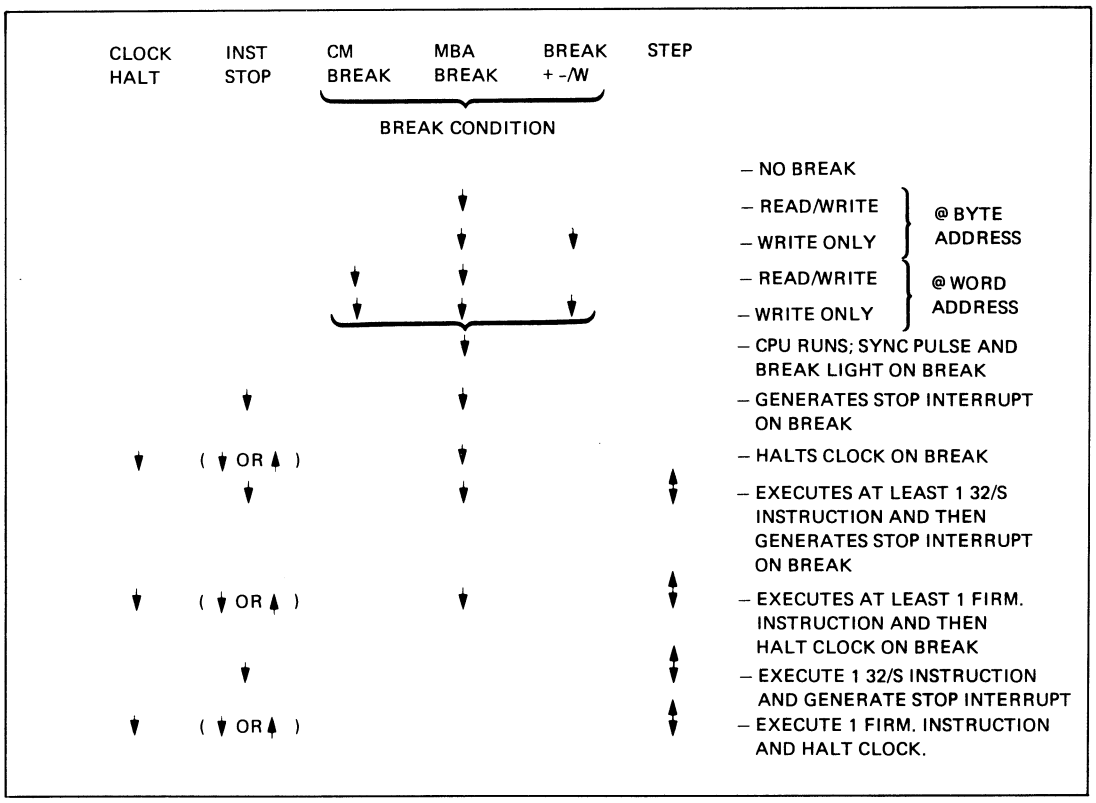


Figure A. Break-Point and Single-Step Operations.

12.5 STATUS INDICATORS

The seven status indicators indicate the dynamic operational state of the computer.

The status indicators are defined as follows:

MEM	:	This indicator is lit if the MOS main memory has power.
ON	:	This indicator is lit if the 32/S computer has power on.
LOCK	:	This indicator is lit if the key lock is in the LOCK position.
CLOCK	:	This indicator is lit if the computer clock is running.
BUS	:	This indicator lights each time a Monobus transfer takes place.
INSTR	:	This indicator lights whenever a 32/S instruction is executed.
BREAK	:	This indicator lights whenever a break condition occurs.

12.6 INITIAL PROGRAM LOAD, IPL

A program can be loaded via the I/O device controller specified by the configuration switches on the Processor Data Board. The IPL is initiated from the LOAD button on the front panel.

A flow chart explaining the Initial Program Load feature is shown in Figure A. Note that the size of the IPL record is device dependent, being a single sector on a disc, a single record on a magnetic tape unit, and a variable-length record on any concurrent I/O device.

The I/O device controller which is to perform the IPL operation is specified by a four-pole switch on the Processor Data board. This switch permits selection of any device number in the range of 0 to 15. The poles of this switch are, reading from left to right:

S3:	bit 3	of device number
S2:	bit 2	
S1:	bit 1	
S0:	bit 0	

The IPL record is read into main memory, starting at absolute location 0. The format of this record must be obtained from Microdata. It includes a brief program segment and data stack.

After the IPL record is loaded, the processor begins execution of the procedure in the record's program segment. This IPL procedure reads in a loader procedure and creates a data stack for use by that procedure. The IPL procedure terminates with a RESM instruction which activates the newly created stack and calls the loader procedure.

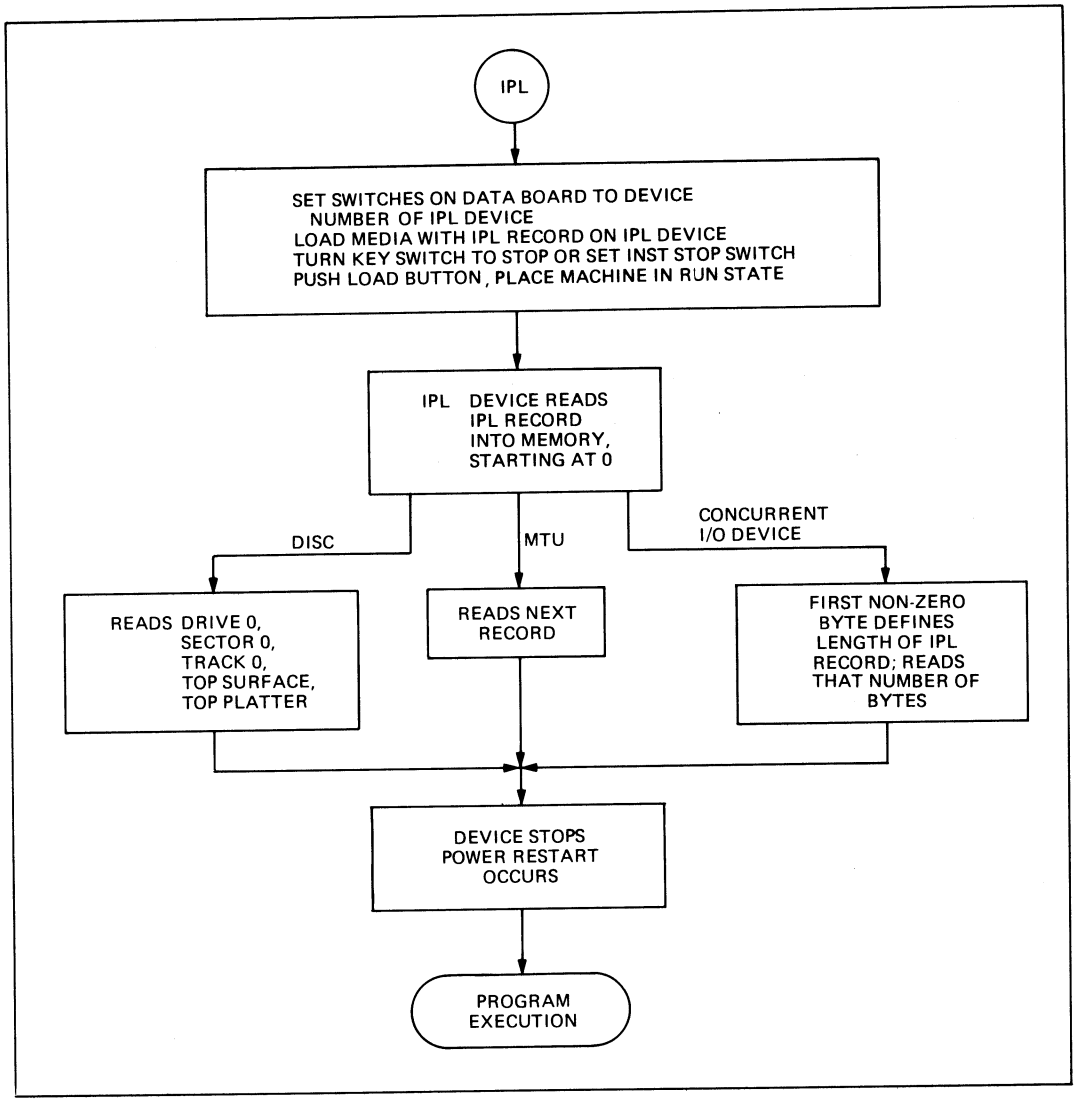


Figure A. Initial Program Load.

13 Power

13.1 POWER REQUIREMENTS

Power requirements are specified for the standard 32/S computer, memory, and I/O device controller modules.

The current drawn from each of the four power supply voltage lines is listed in Figure A. Each system requirement must be checked for current requirements, particularly on the +5 volt line before being assembled.

The power to 32/S modules is supplied via the backplane. Normally all power is provided by the integral power supply. However, when additional power is required, it can be provided by using an additional power supply.

An additional power supply can be used to supply additional +5 volt current. To do this, two cuts must be made in the +5 volt printed circuit etch of the backplane. This will separate the +5 volt line used by the front card slots from the +5 volt line used by the back card slots. The cuts can be made between any pair of adjacent slots starting with the seventh and eighth. A remote-mounted supply is then plugged into the rear end of the backplane.

Standard rack-mounted power supplies are available to provide 20 amps or to supply 40 amps on the +5 volt line.

Instructions for cutting the backplane, ordering, and installing the additional power supply may be obtained from Microdata.

NOTE: The user must keep in mind the power requirements of various modules if he wishes to move modules around in a system configured with an additional remote power supply.

MODULE	+5.0V	+12V	-12V	+21V	+5.3V
DATA } CPU	4.0	–	–	–	–
MONOBUS INTERFACE } CPU	2.5	–	–	–	–
PROCESSOR CONTROL } CPU	5.0	–	–	–	–
8K X 16 MOS MEMORY – ACTIVE	0.35	–	–	0.5	0.7
– INACTIVE	0.35	–	–	0.2	0.7
– STANDBY	0	–	–	0.2	0.02
MAINTENANCE FRONT PANEL (SWITCH AND LOGIC)	5.6	–	–	–	–
BASIC FRONT PANEL	0.1	–	–	–	–
READ-ONLY CONTROL MEMORY EXPANSION, 512 WORDS	2.5	–	–	–	–
READ-ONLY CONTROL MEMORY EXPANSION, 1K WORDS	3.0	–	–	–	–
WRITABLE CONTROL MEMORY, 512 WORDS	7.4	–	–	–	–
MPI/O CONTROLLER	3.8			–	–
DISC CONTROLLER	4.6	–	–	–	–
MAGNETIC TAPE CONTROLLER	3.8	–	–	–	–
BUS COUPLER	1.0	–	–	–	–
TOTAL AVAILABLE FROM INTEGRAL POWER SUPPLY	35.0A	3.0A	3.0A	4.0A	6.0A

Figure A. Module Power Requirements.

13.2 POWER FAIL & RESTART

The software servicing of a power fail interrupt and firmware reaction to a restart are described.

The power fail circuitry in the power supply provides a power fail interrupt (interrupt vector number 0) when a loss of AC power is detected. A minimum of 2 milliseconds is assured from that time until the DC power drops below regulation. The processor will go into the run state if it had been in the wait state.

The Interrupt processing procedure must be executed in the interrupt stack (S = 1 in the power fail interrupt vector). The last instruction after preparing for the power fail must be a WAIT instruction. This instruction builds a Mark in the interrupt stack.

When power is restored, the processor restarts execution at the instruction following the WAIT instruction, operating out of the interrupt stack.

A restart occurs when the key lock is switched from OFF or HOLD to STOP, after restoration of power following a power failure, or after an initial program load (see Initial Program Load, topic 12.6). The Interrupt Stack Descriptor (locations "00" and "02") is used to set up the SB and SL registers. The contents of the word addressed by SB are transferred to the SP register. The word at SP-4 is stored into the EP register. The word at SP is used to restore the PSR and then PLIB is accessed to set up PB and PL. PP is restored from the word at SP-2 and then the top four words of the stack are popped (SP is decremented by 8). The interrupt stack active status bit is set and the wait status bit is reset. Instruction execution starts if and when the key switch is set in the RUN or LOCK position.

A I/O Device Controllers

A-1 SERIAL INPUT, MPI/O CONTROLLER

The Device Register Block is defined for the serial input controller in the MPI/O module.

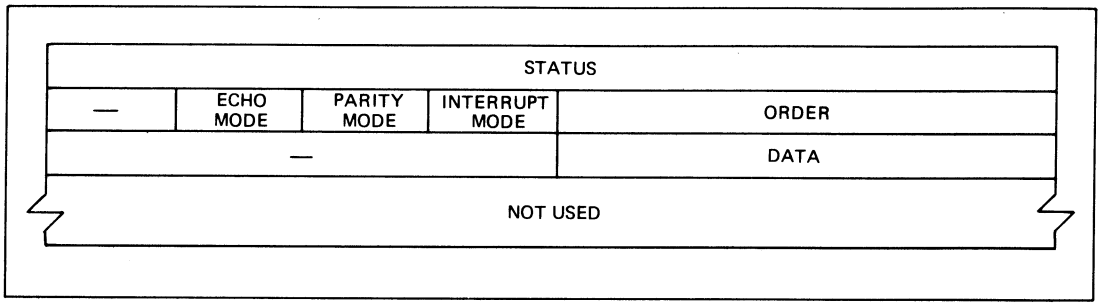


Figure A. Device Register Block, Serial Input, MPI/O.

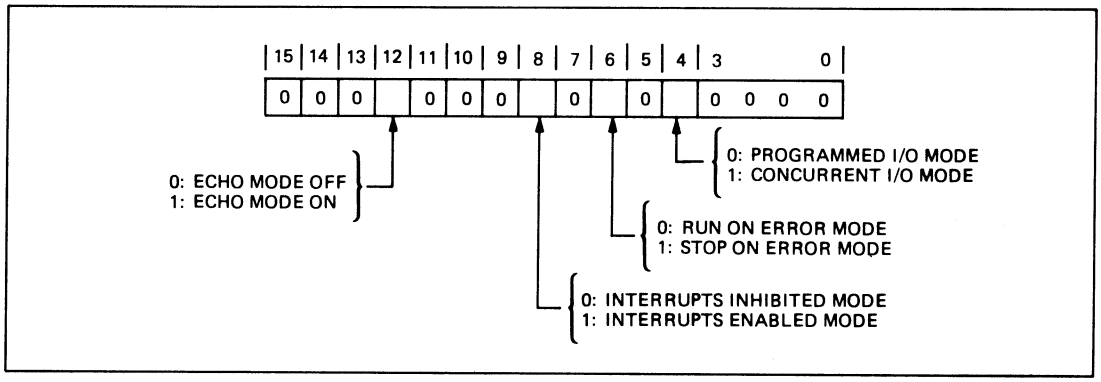


Figure B. Read Format of Mode and Order Fields, Serial Input, MPI/O.

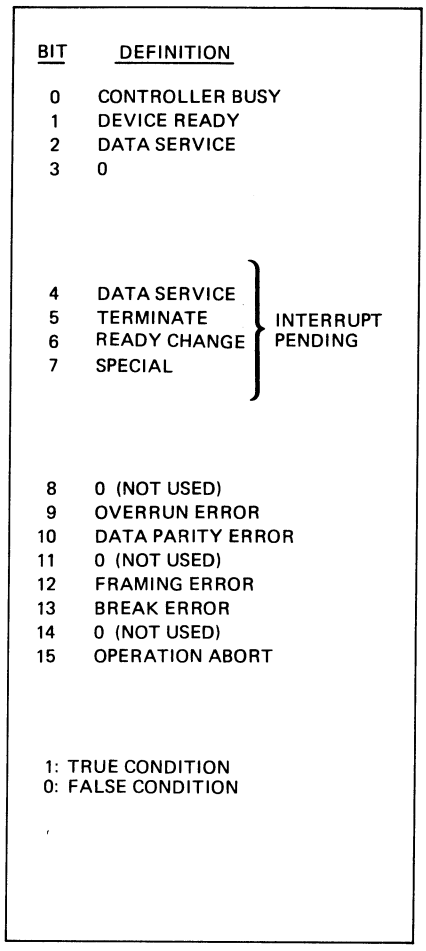


Figure C. Status Field, Serial Input MPI/O.

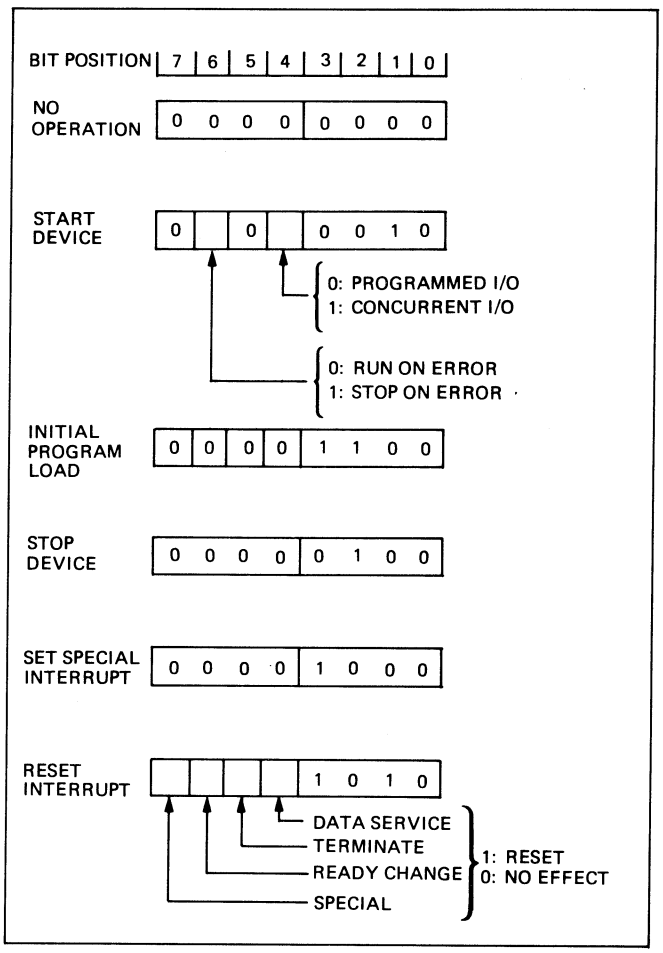


Figure D. Order Field, Serial Input, MPI/O.

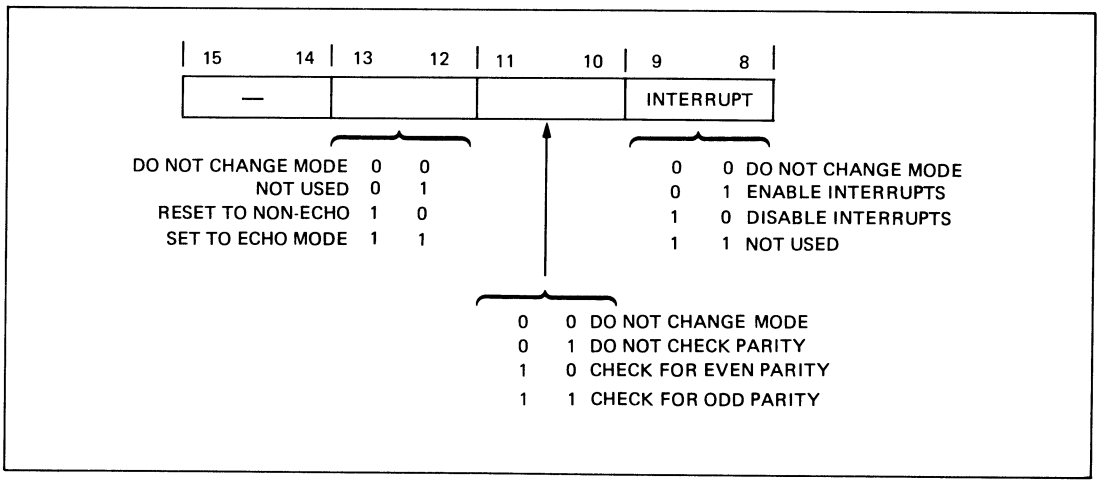


Figure E. Mode Field, Serial Input, MPI/O.

A-2 SERIAL OUTPUT, MPI/O CONTROLLER

The Device Register Block is defined for the serial output controller in the MPI/O module.

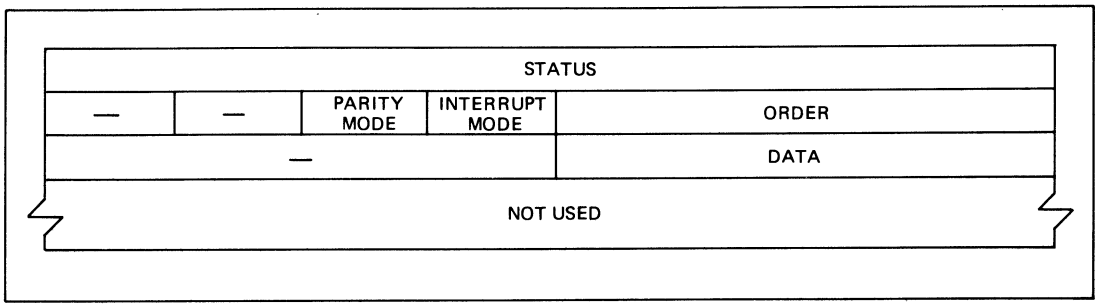


Figure A. Device Register Block, Serial Output, MPI/O.

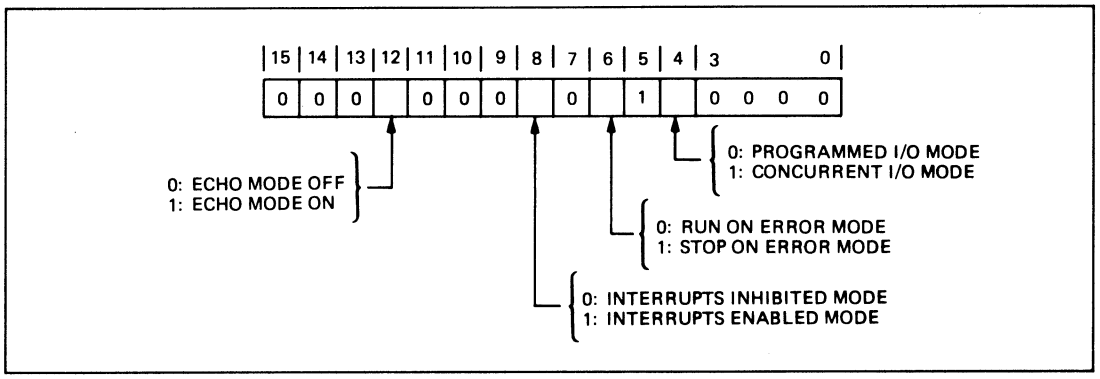


Figure B. Read Format of Mode and Order Fields, Serial Output, MPI/O.

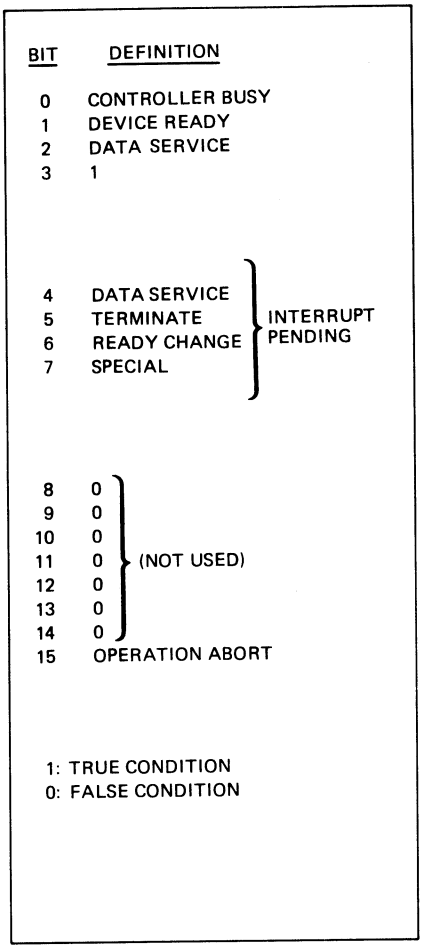


Figure C. Status Field, Serial Output, MPI/O.

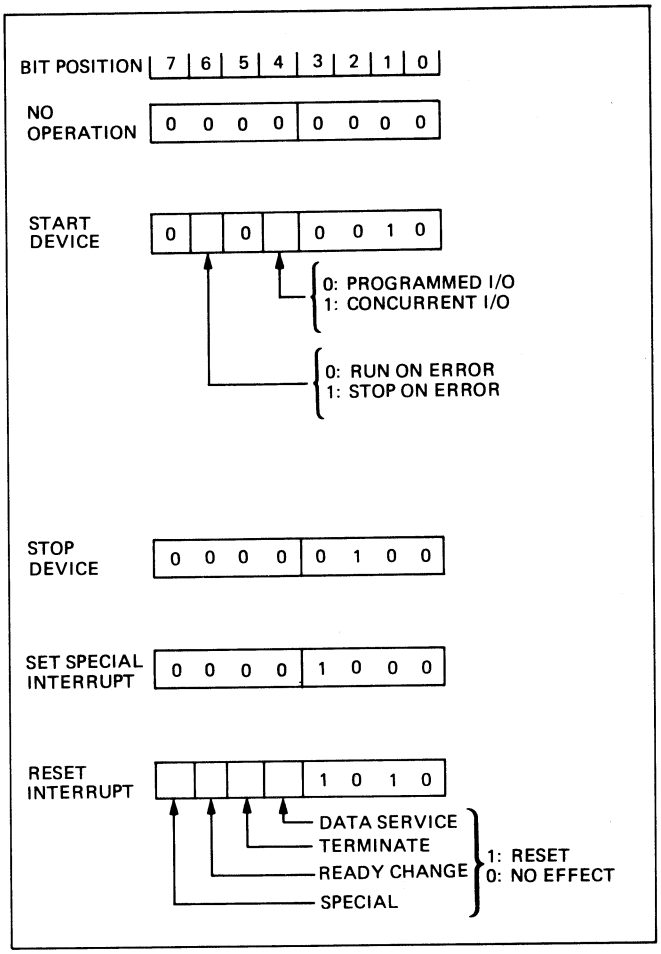


Figure D. Order Field, Serial Output, MPI/O.

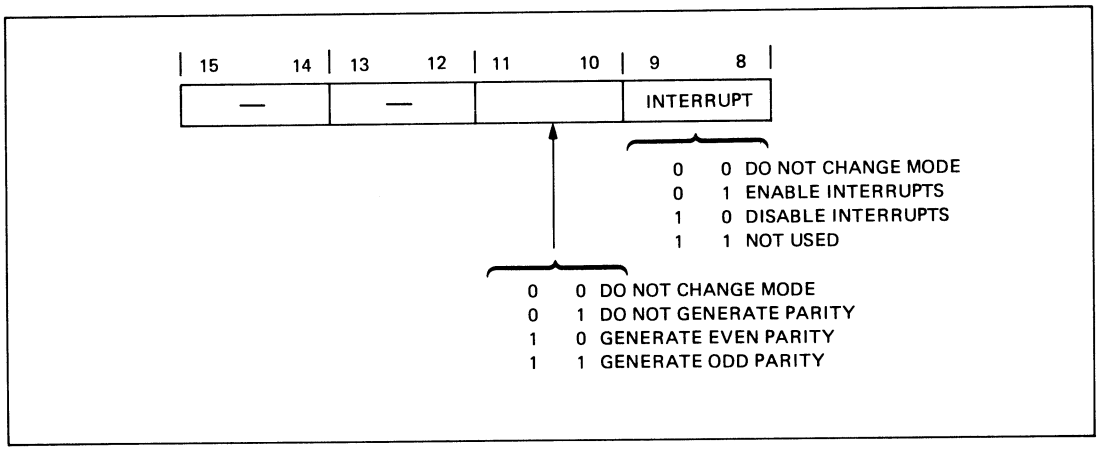


Figure E. Mode Field, Serial Output, MPI/O.

A-3 PAPER TAPE READER, MPI/O CONTROLLER

The Device Register Block is defined for the parallel input controller in the MPI/O module, when used as a paper tape reader controller.

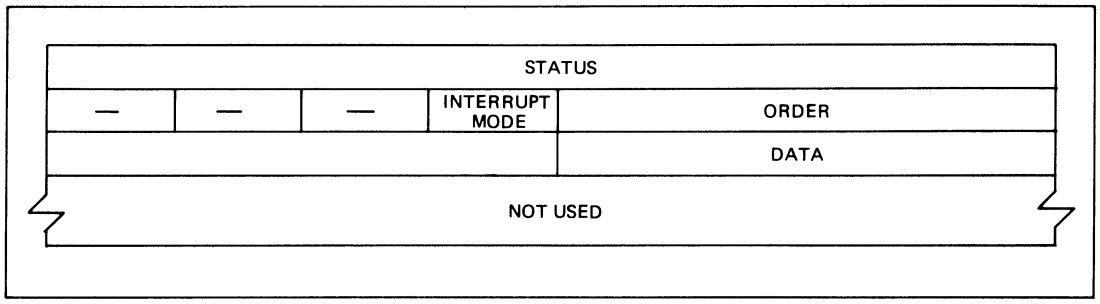


Figure A. Device Register Block, Paper Tape Reader, MPI/O.

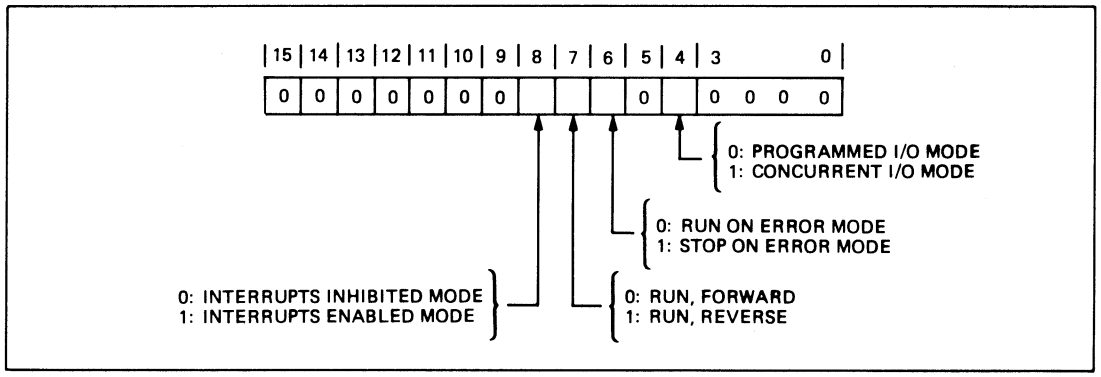


Figure B. Read Format of Mode and Order Fields, Paper Tape Reader, MPI/O.

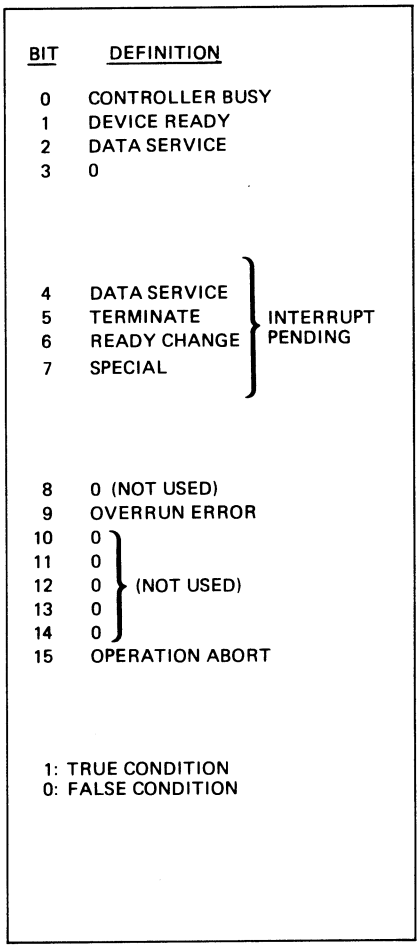


Figure C. Status Field, Paper Tape Reader, MPI/O.

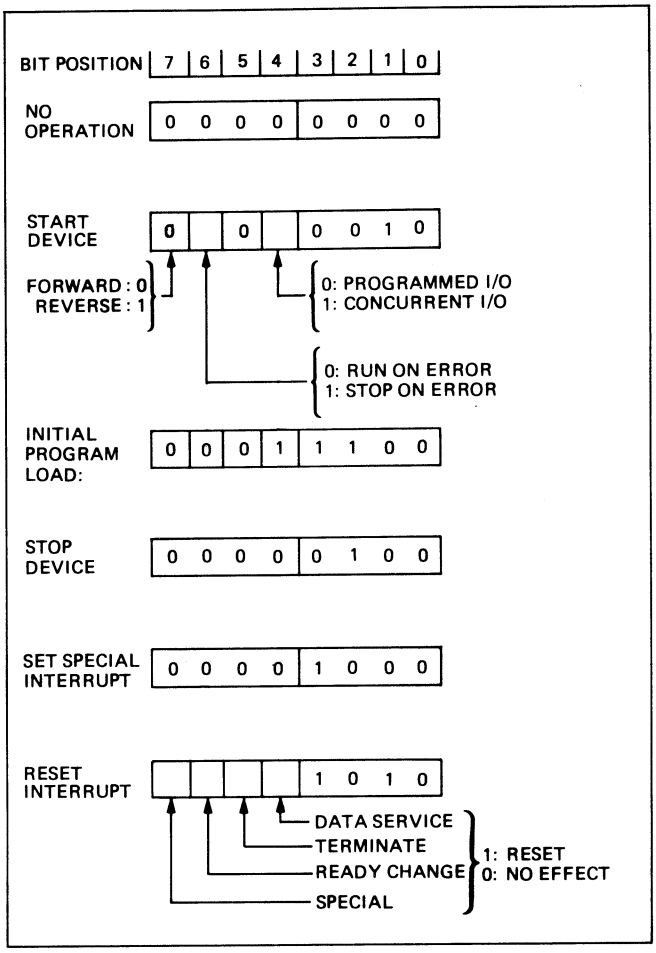


Figure D. Order Field, Paper Tape Reader, MPI/O.

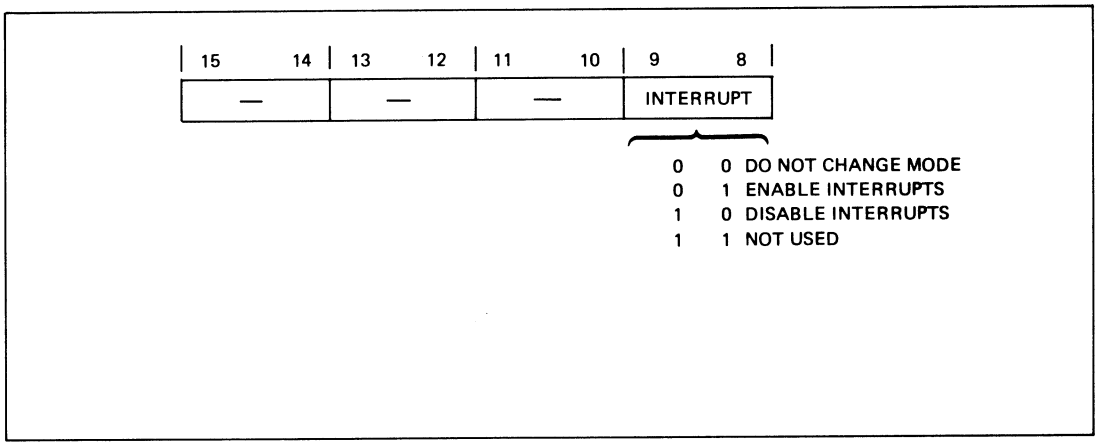


Figure E. Mode Field, Paper Tape Reader, MPI/O.

A-4 PAPER TAPE PUNCH, MPI/O CONTROLLER

The Device Register Block is defined for the parallel output controller in the MPI/O module, when used as a paper tape punch controller.

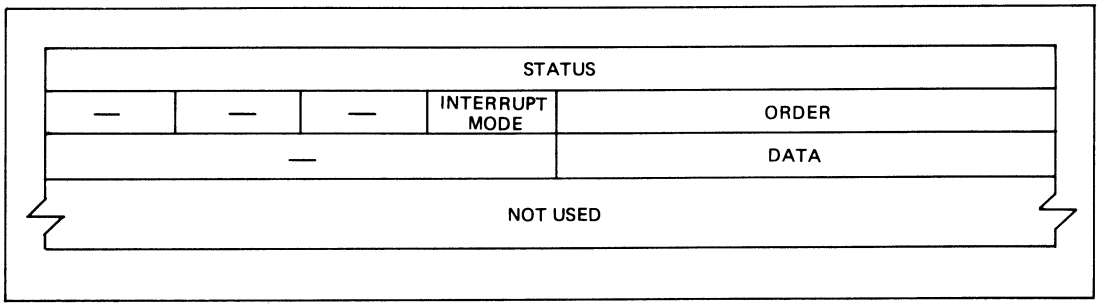


Figure A. Device Register Block, Paper Tape Punch, MPI/O.

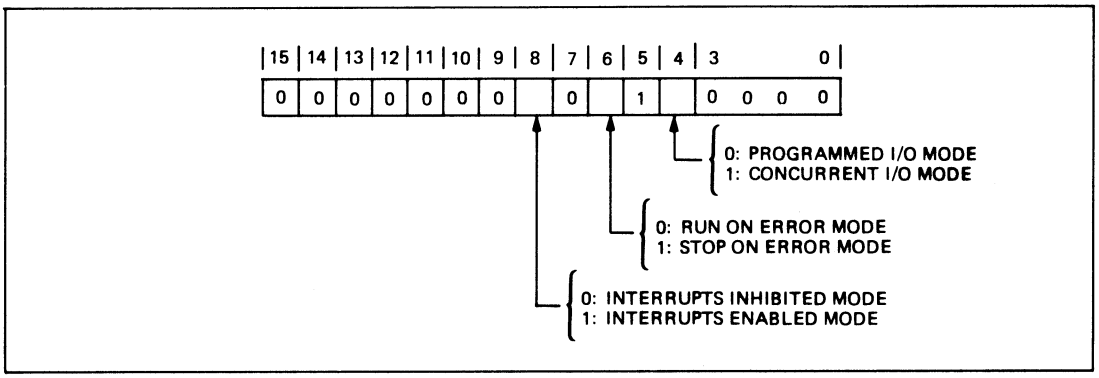


Figure B. Read Format of Mode and Order Fields, Paper Tape Punch, MPI/O.

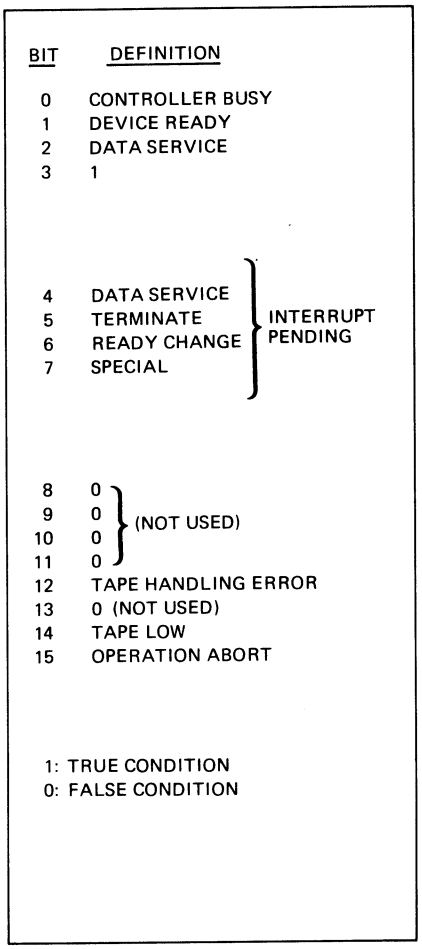


Figure C. Status Field, Paper Tape Punch, MPI/O.

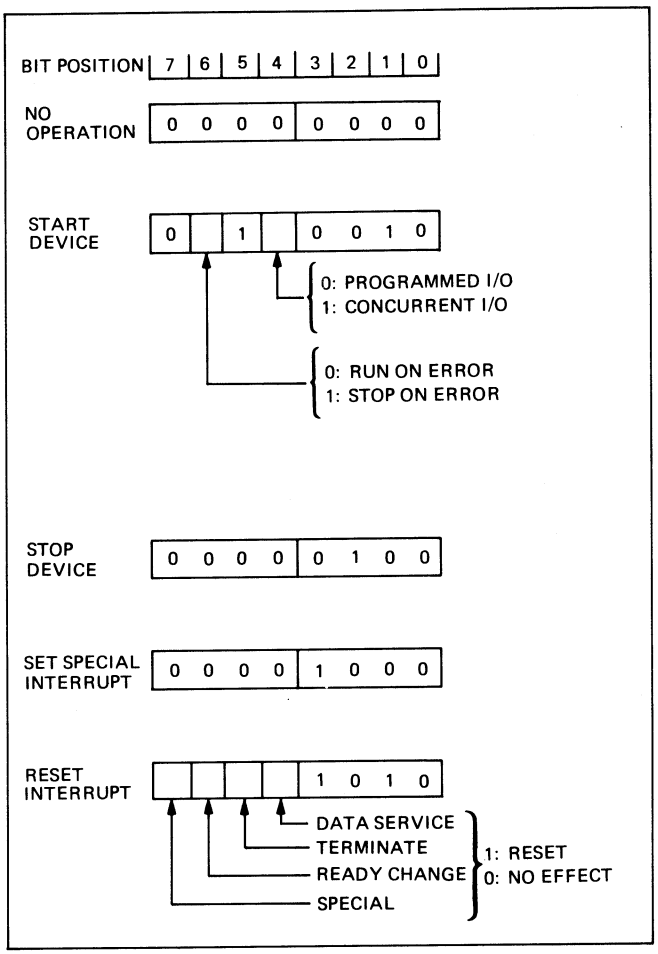


Figure D. Order Field, Paper Tape Punch, MPI/O.

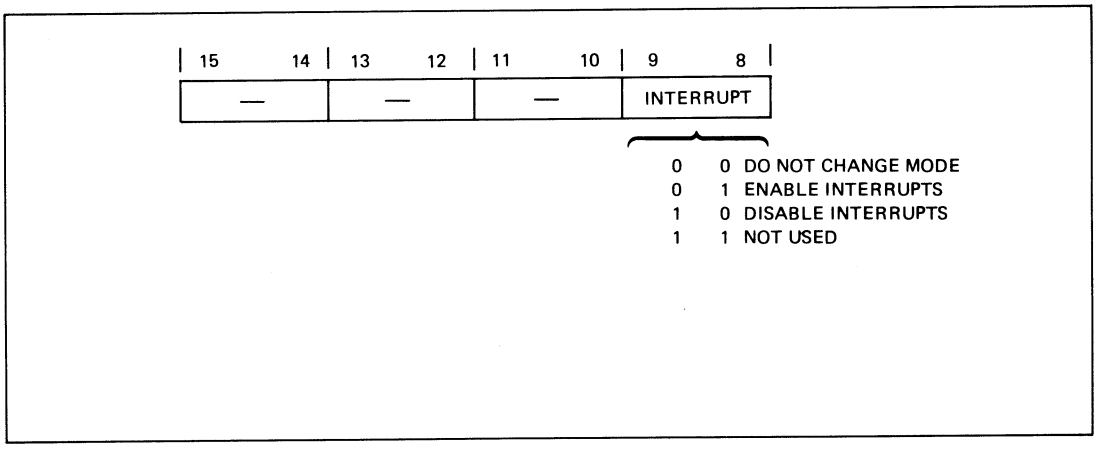


Figure E. Mode Field, Paper Tape Punch, MPI/O.

A-5 CARD READER, MPI/O CONTROLLER

The Device Register Block is defined for the parallel input controller in the MPI/O module, when used as a card reader controller.

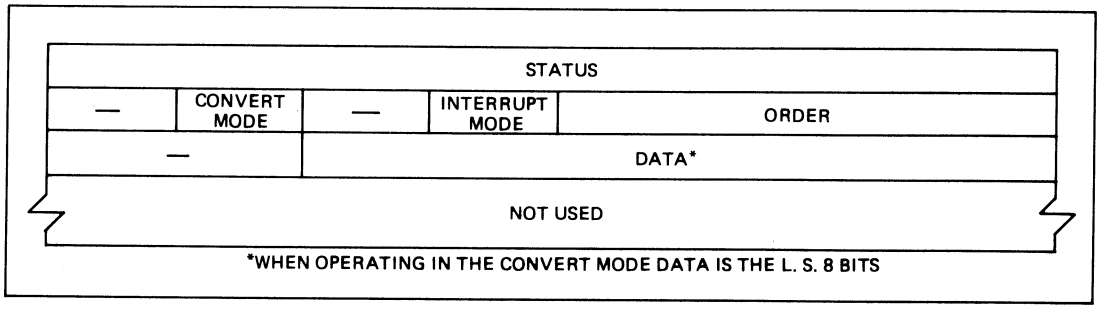


Figure A. Device Register Block, Card Reader, MPI/O.

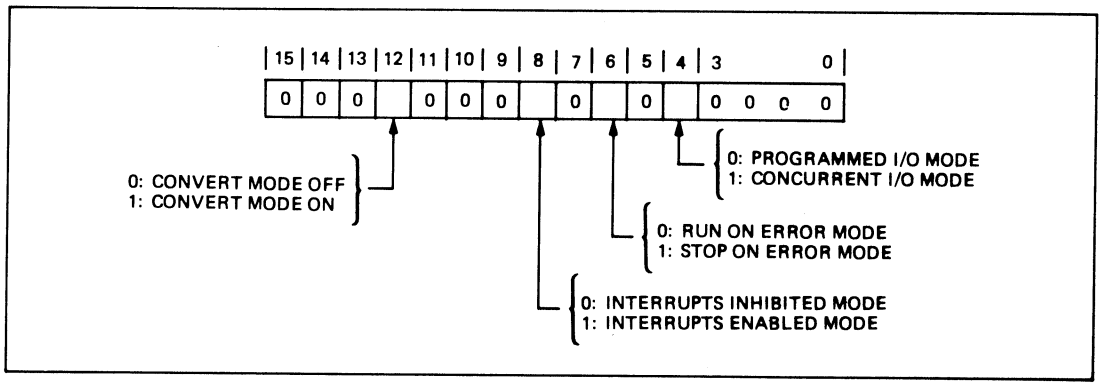


Figure B. Read Format of Mode and Order Fields, Card Reader, MPI/O.

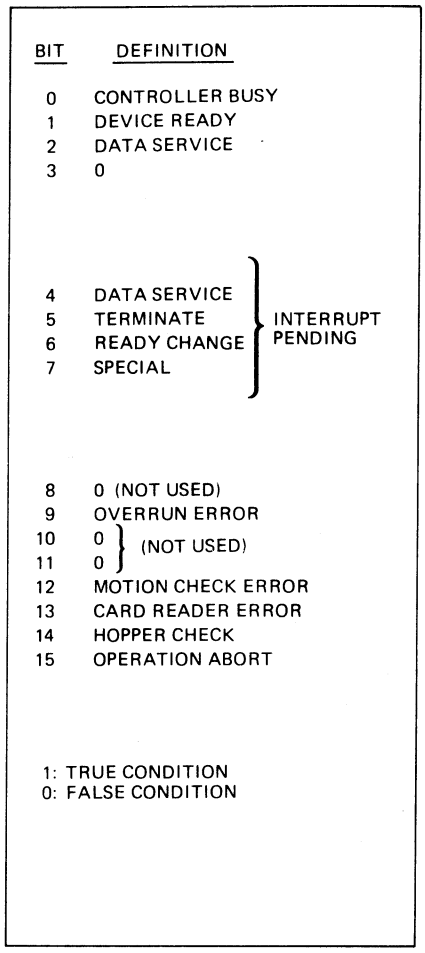


Figure C. Status Field, Card Reader, MPI/O.

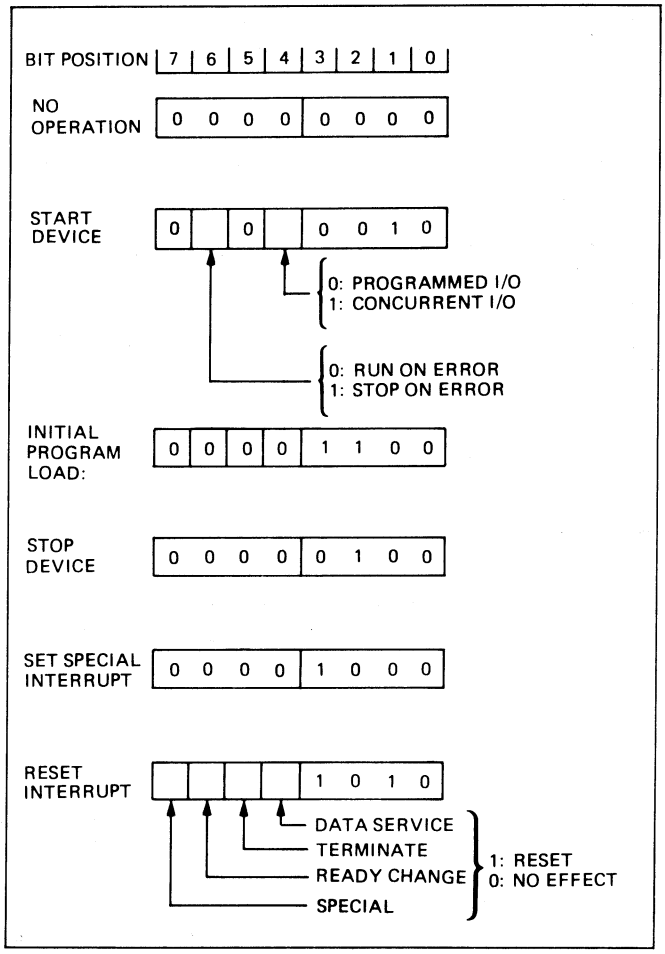


Figure D. Order Field, Card Reader, MPI/O.

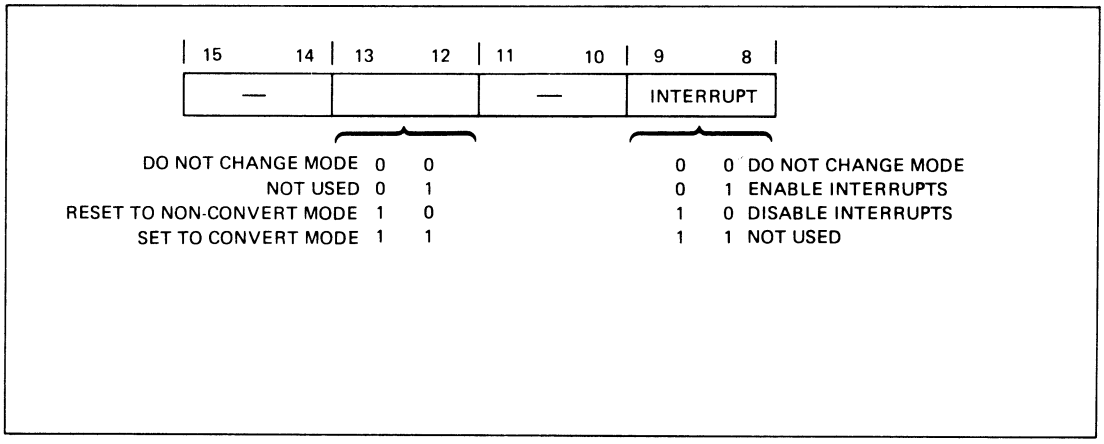


Figure E. Mode Field, Card Reader MPI/O.

A-6 LINE PRINTER, MPI/O CONTROLLER

The Device Register Block is defined for the parallel output controller in the MPI/O module, when used as a line printer controller.

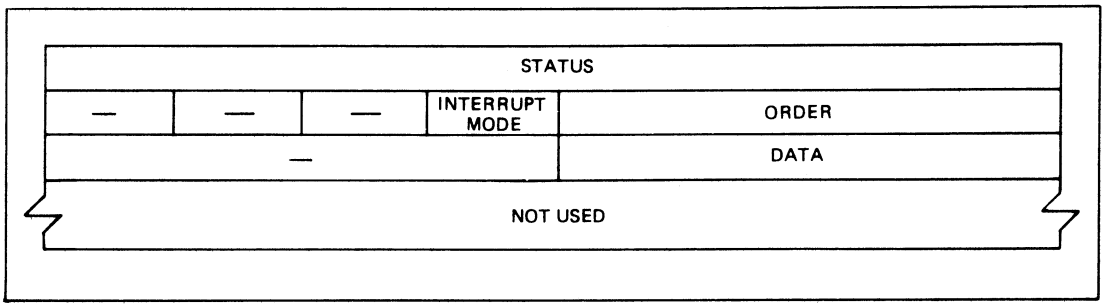


Figure A. Device Register Block, Line Printer, MPI/O.

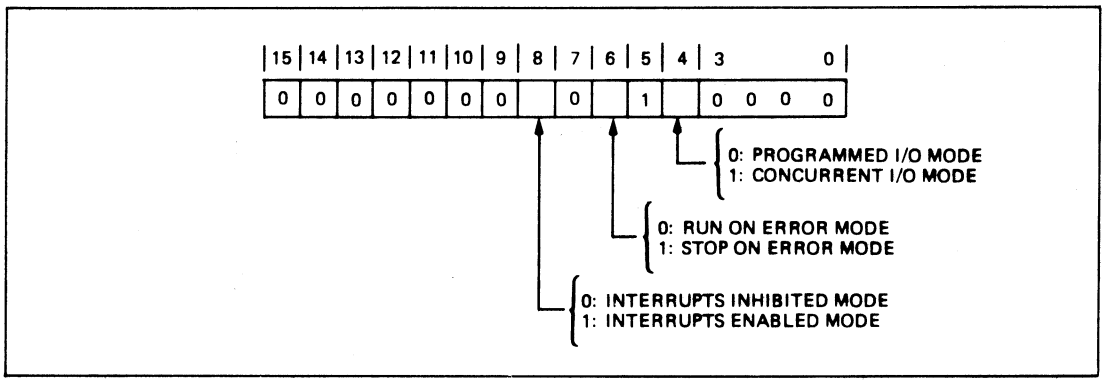


Figure B. Read Format of Mode and Order Fields, Line Printer, MPI/O.

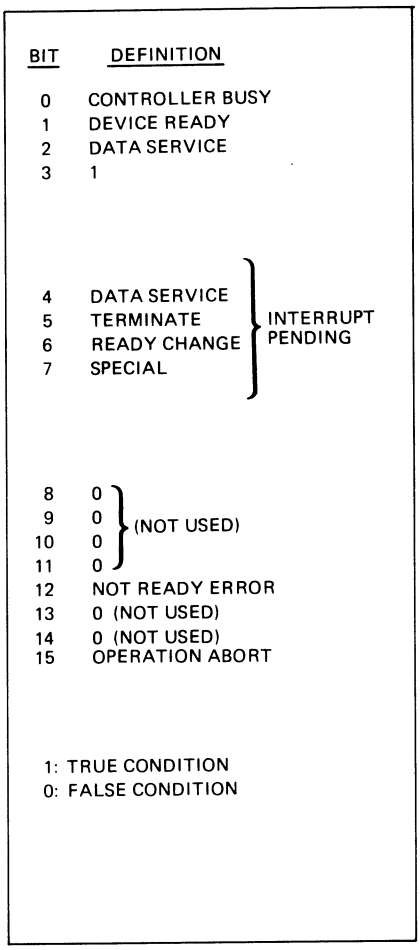


Figure C. Status Field, Line Printer, MPI/O.

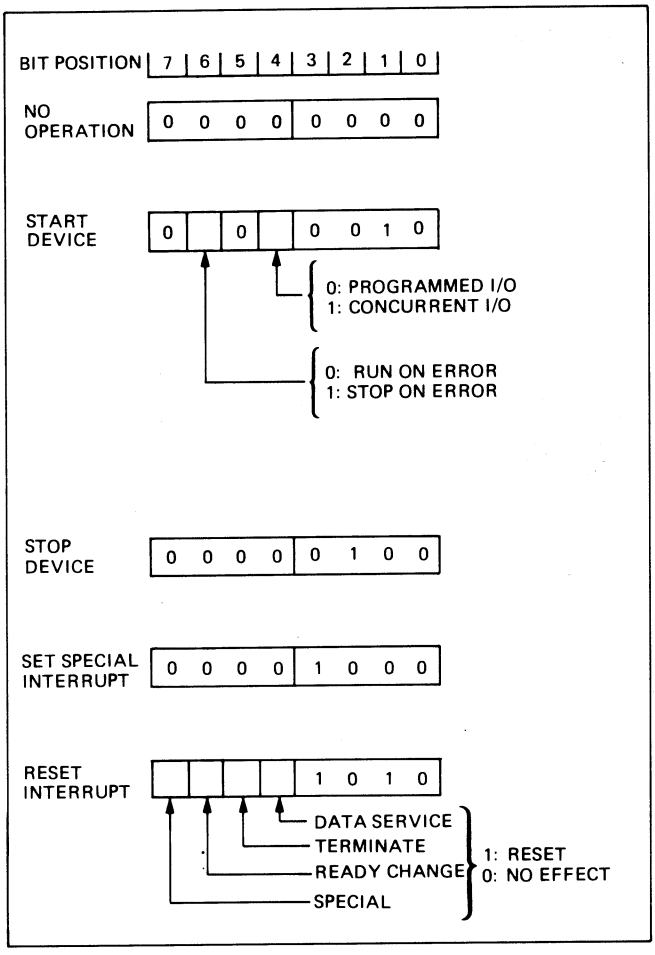


Figure D. Order Field, Line Printer, MPI/O.

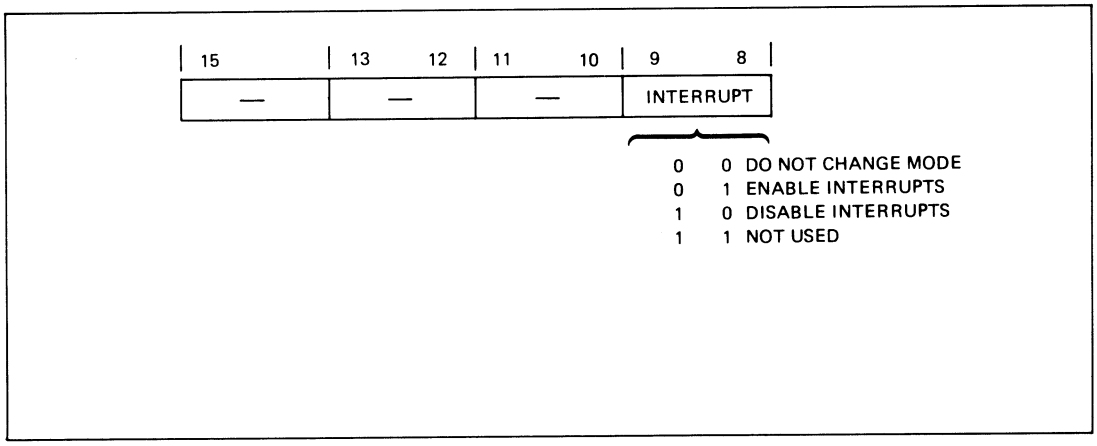


Figure E. Mode Field, Line Printer, MPI/O.

A-7 DISC CONTROLLER

The Device Register Block is defined for the Disc controller. This topic covers the status, order, and mode fields.

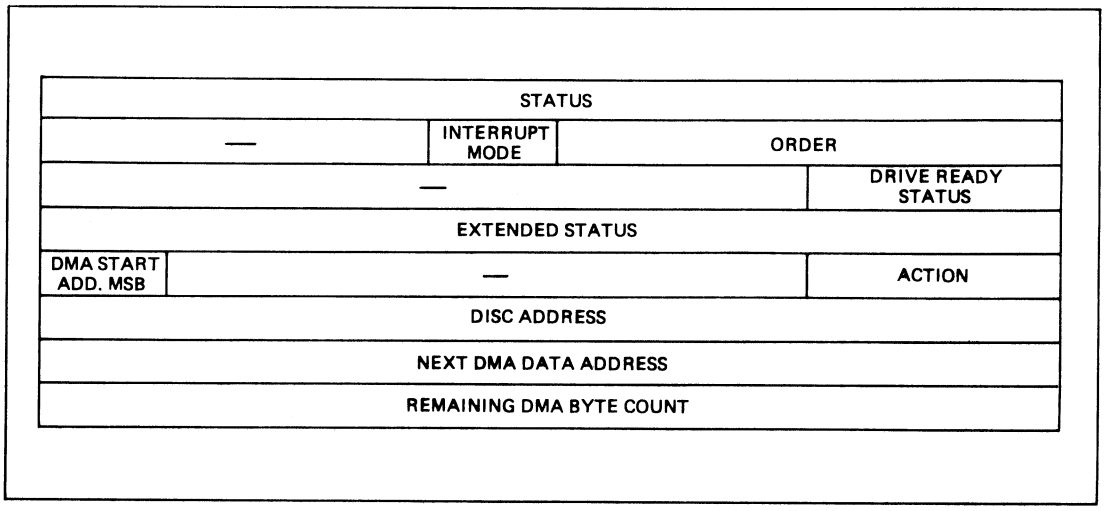


Figure A. Device Register Block, Disc.

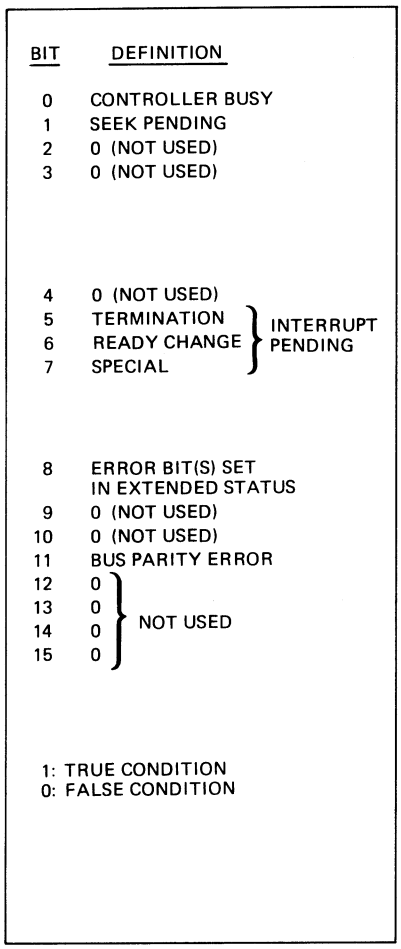


Figure B. Status Field, Disc.

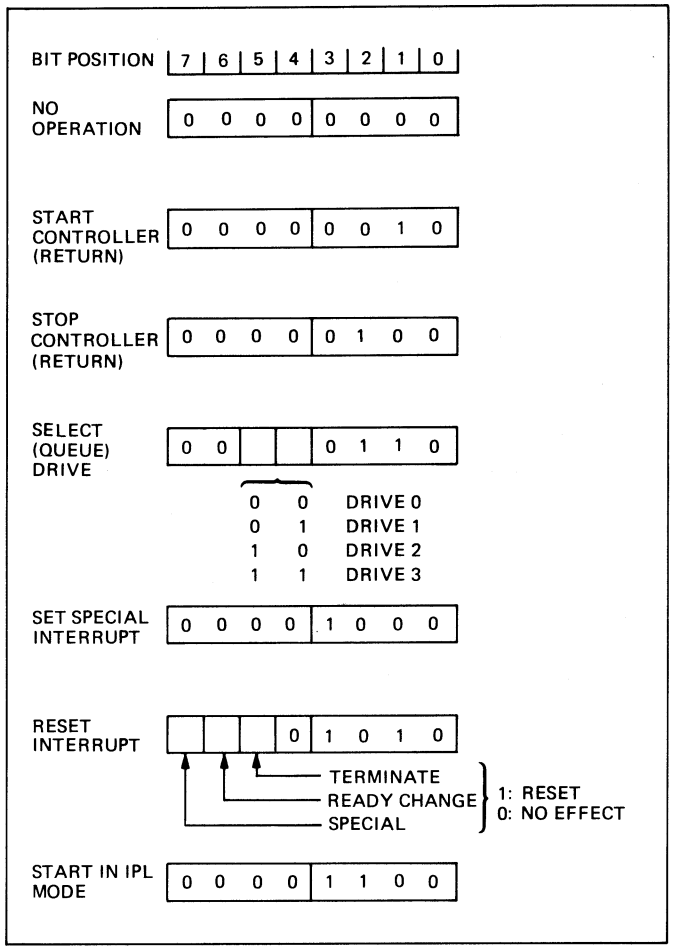


Figure C. Order Field, Disc.

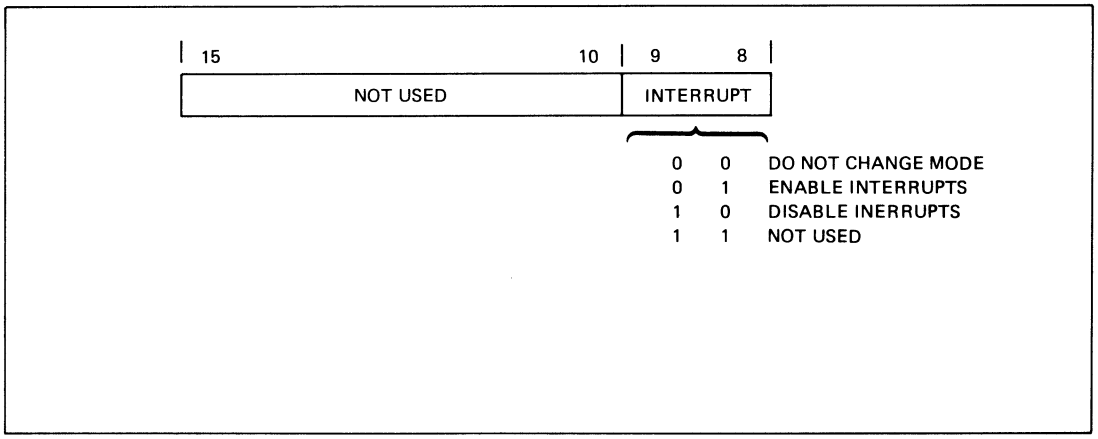


Figure D. Mode Field, Disc.

A-8 DISC CONTROLLER (Continued)

The Device Register Block is defined for the Disc controller. This topic covers the extended status, drive ready status, action, and disc address fields.

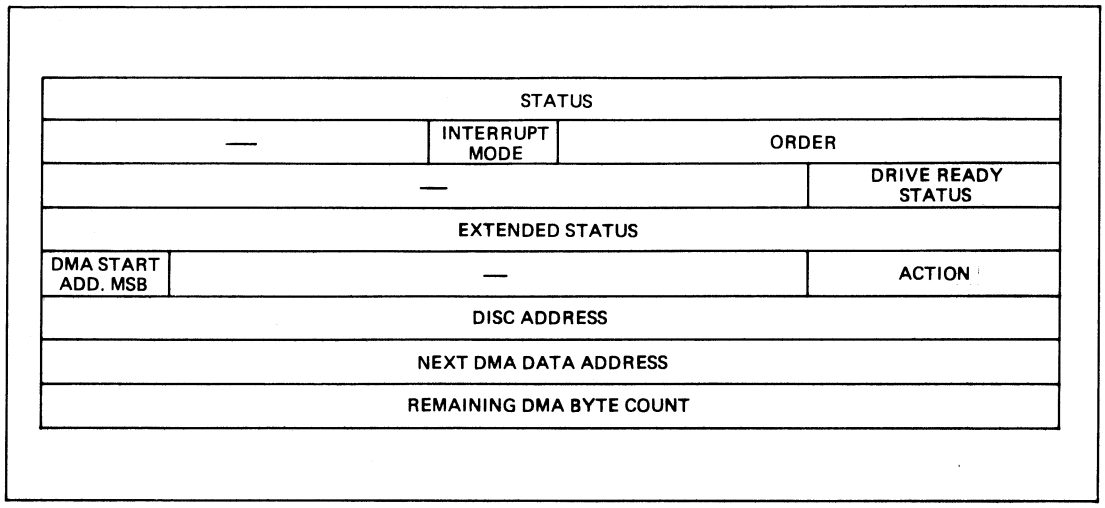


Figure A. Device Register Block, Disc.

BIT		DEFINITION
0	0 0	DRIVE NO. 0
1	0 1	DRIVE NO. 1
	1 0	DRIVE NO. 2
	1 1	DRIVE NO. 3
2	0 (NOT USED)	
3	0 (NOT USED)	
4	NOT READY ERROR (DRIVE QUEUED, CONTROLLER STARTED, DRIVE IS NOT READY)	
5	NO SEEK ERROR (CONTROLLER STARTED, NO DRIVE QUEUED)	
6	SEEK INCOMPLETE ERROR (TRACK NOT FOUND)	
7	SECTOR NUMBER ERROR (SECTOR NOT FOUND,>23)	
8	PLATTER FORMAT PROTECT ERROR	
9	SECTOR WRITE PROTECT ERROR	
10	DISC ADDRESS ERROR	
11	HEADER CYCLIC REDUNDANCY CHECK CODE ERROR	
12	DMA OVERRUN ERROR	
13	DATA CYCLIC REDUNDANCY CHECK CODE ERROR	
14	0 (NOT USED)	
15	0 (NOT USED)	

1: TRUE CONDITION
0: FALSE CONDITION

Figure B. Extended Status Field, Disc.

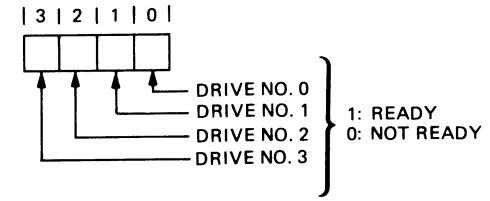


Figure C. Drive Ready Status Field, Disc.

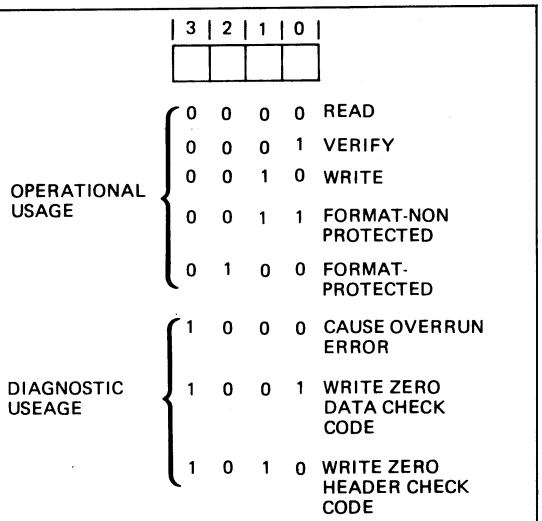
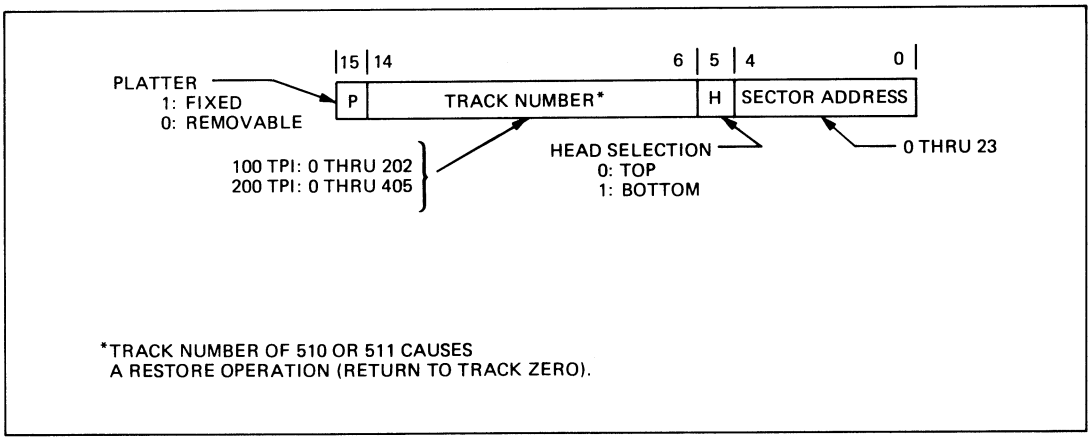


Figure D. Action Field, Disc.



*TRACK NUMBER OF 510 OR 511 CAUSES A RESTORE OPERATION (RETURN TO TRACK ZERO).

Figure E. Disc Address Field, Disc.

A-9 MAGNETIC TAPE FORMATTER CONTROLLER

The Device Register Block is defined for the Magnetic Tape Formatter controller. This topic covers the status, order (controller orders), and mode fields.

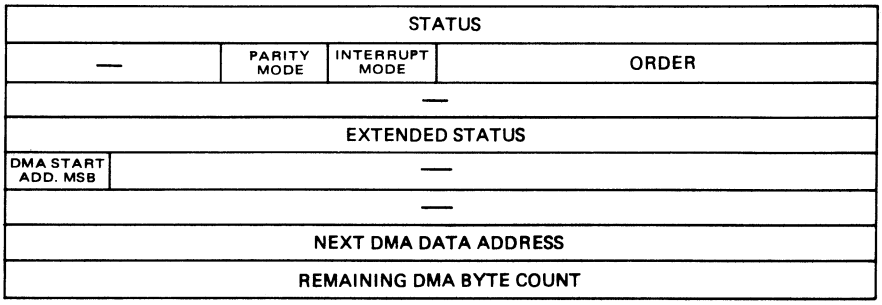


Figure A. Device Register Block, Magnetic Tape Formatter Controller

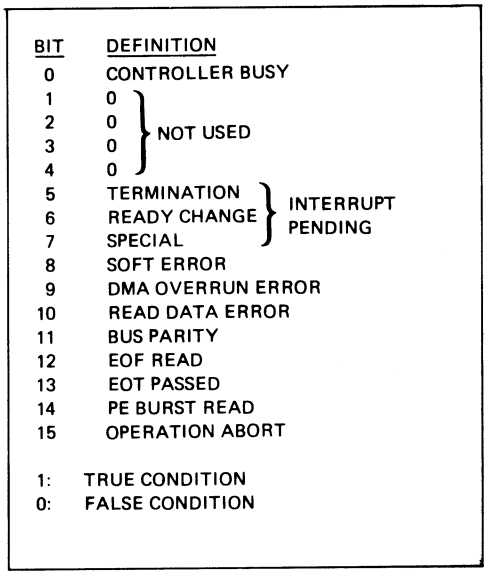


Figure B. Status Field, Magnetic Tape

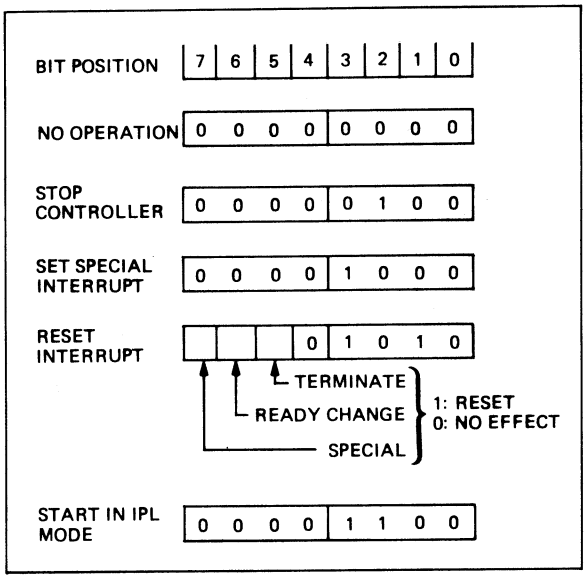


Figure C. Order Field, Magnetic Tape (Controller Orders)

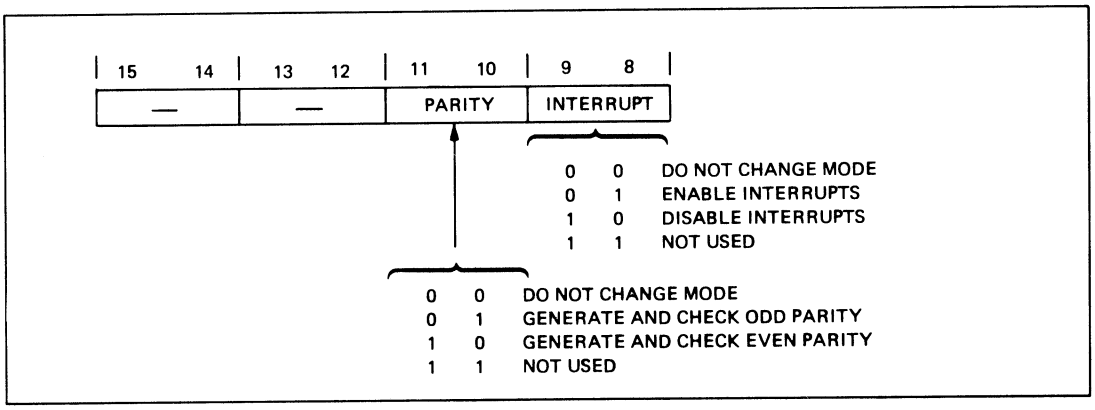


Figure D. Mode Field, Magnetic Tape

A-10 MAGNETIC TAPE FORMATTER CONTROLLER (Continued)

The Device Register Block is defined for the Magnetic Tape Formatter controller. This topic covers the extended status and order (drive select orders) fields.

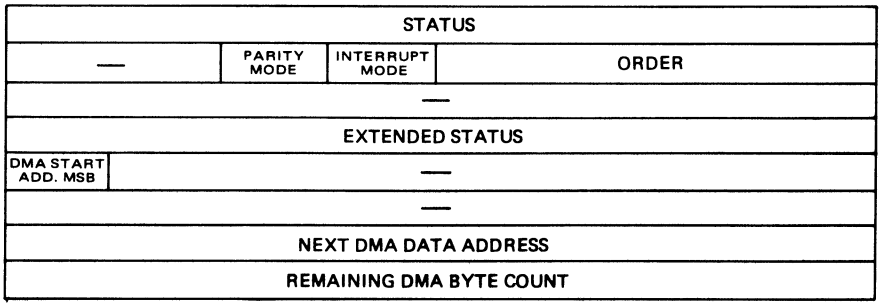


Figure A. Device Register Block, Magnetic Tape Formatter Controller

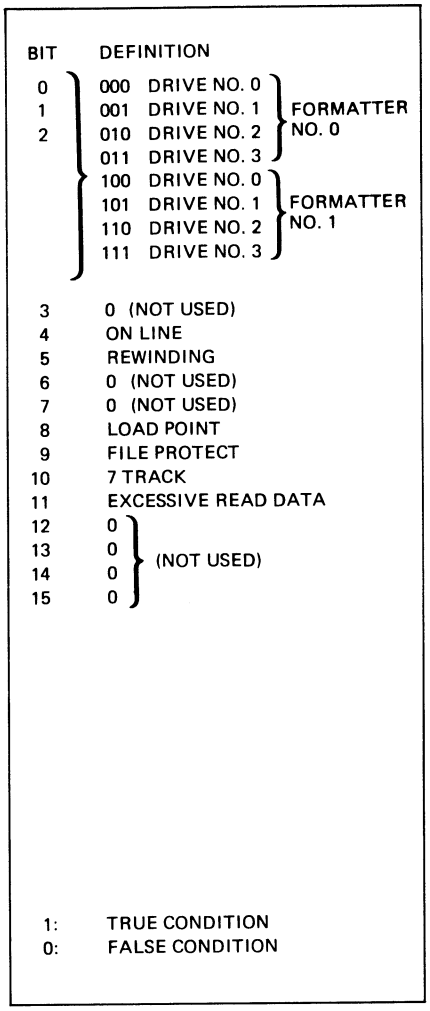


Figure B. Extended Status Field, Magnetic Tape

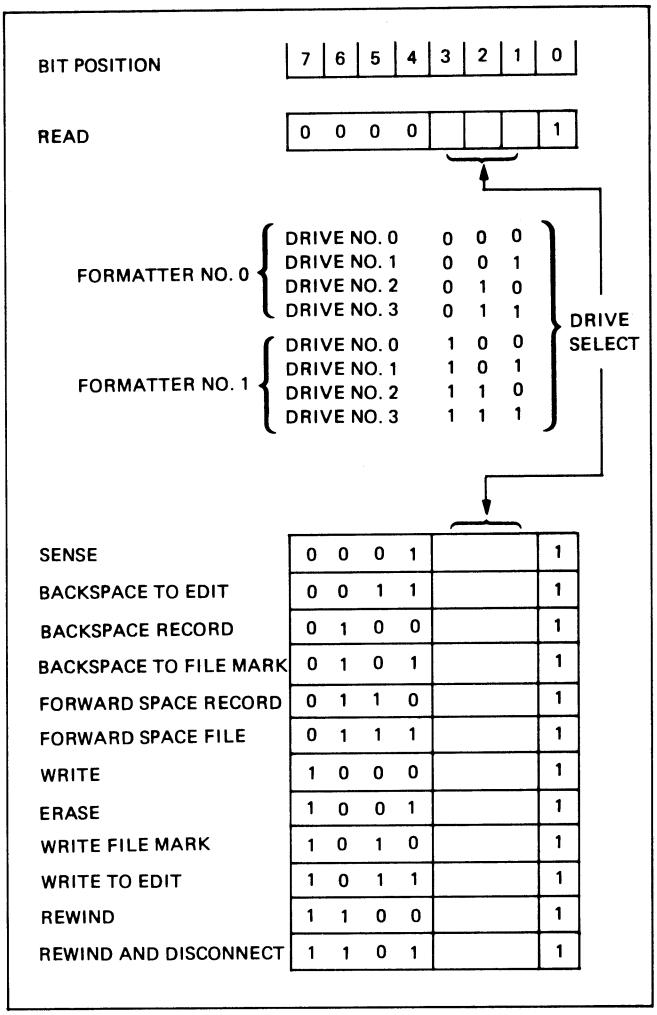


Figure C. Order Field, Magnetic Tape (Drive Select Orders)

B. Index and Tables

B.1 INDEX TO REGISTERS AND FORMATS

An index is provided of registers and formats.

		Topic
PB	Program Base	2.2
PL	Program Length	2.2
PP	Program Pointer	2.2
PRT	Program Reference Table	2.2
PRTN	Program Reference Table Number	2.2
PLIBN	Program Library	2.3
SB	Stack Base	2.5
SL	Stack Length	2.5
EP	Environmental Pointer	2.5
SP	Stack Pointer	2.5
TOS	Top of Stack Registers	2.5
TOS1		2.5
TOS2		2.5
TOS3		2.5
TOS4		2.5
--	Stack Head Registers	2.5
--	Procedure Mark	2.8
--	Begin Mark	2.8
--	Interrupt Mark	2.9
DLEX	Delta Lex Level	2.11
PSR	Program Status Register	2.16
--	Interrupt Vector Table	3.2
--	Interrupts	3.3
DRB	Device Register Block	4.2
--	Controller Response Word	4.3
CCIOB	Concurrent I/O Control Block	4.5
--	Field Descriptor	7.11
--	String Descriptor	10.1
--	Word Index	7.12
--	Array Index	7.12

B.2 INDEX TO INSTRUCTIONS

An index to instructions is provided.

		Op Code	Topic
MEMORY REFERENCE			
Addressing Modes	**Mode**		
Global Direct	0		6.2
Global Direct, Indexed	1		6.2
Local Direct	2		6.2
Local Direct, Indexed	3		6.2
Indirect Thru TOS	4		6.2
Indirect Thru TOS, Indexed	5		6.2
Constant Direct, Indexed	6		6.2
Absolute, Indexed	7		6.2
STF	Store Field	1	6.3
STB	Store Byte	7	6.3
STW	Store Word	6	6.3
STWN	Store Word, Non-Destructive	5	6.3
STD	Store Doubleword	0	6.3
SST	Store Tripleword	3	6.3
LF	Load Field	A	6.4
LB	Load Byte	F	6.4
LW	Load Word	E	6.4
LD	Load Doubleword	8	6.4
LTW	Load Tripleword	D	6.4
AWM	Add Word to Memory	4	6.5
AW	Add Word to Stack	C	6.5
SW	Subtract Word from Stack	D	6.5
SWAP	Swap Word in Stack with Memory	2	6.6
STACK OPERATE			
ADD	Add	20	7.2
SUB	Subtract	21	7.2
NEG	Negate	10	7.2
ABS	Absolute Value	1E	7.2
MUL	Integer Multiply	22	7.2
MULD	Multiply with Doubleword Product	36	7.2
DIV	Integer Word Divide	23	7.2
MOD	Module	24	7.2
DADD	Doubleword Add	4F00	7.3
DSUB	Doubleword Subtract	4F01	7.3
DNEG	Doubleword Negate	3C	7.3
DABS	Doubleword Absolute Value	3E	7.3
DMUL	Doubleword Integer Multiply	4F02	7.3
DDIV	Doubleword Integer Divide	4F03	7.3
DIVD	Divide Doubleword by Word	4F14	7.3
DMOD	Doubleword Modulo	4F04	7.3
MODD	Doubleword Modulo by Word	4F15	7.3
FADD	Floating Point Add	4F20	7.4
FSUB	Floating Point Subtract	4F21	7.4
FMUL	Floating Point Multiply	4F22	7.4
FDIV	Floating Point Divide	4F23	7.4
FABS	Floating Point Absolute Value	4F18	7.4

STACK OPERATE (Cont'd)		Op Code	Topic
MAX	Maximum Value	34	7.5
MIN	Minimum Value	35	7.5
DMAX	Doubleword Maximum Value	4F0E	7.5
DMIN	Doubleword Minimum Value	4F0F	7.5
FMAX	Floating Point Maximum Value	4F2E	7.5
FMIN	Floating Point Minimum Value	4F2F	7.5
SGN	Sign Value	44	7.5
DSGN	Doubleword Sign Value	45	7.5
FSGN	Floating Point Sign Value	4F19	7.5
AND	Logical AND	25	7.6
NOT	Logical Not	11	7.6
OR	Logical OR	26	7.6
XOR	Logical EXCLUSIVE OR	27	7.6
DAND	Doubleword Logical AND	4F05	7.6
DNOT	Doubleword Logical Not	3D	7.6
DOR	Doubleword Logical OR	4F06	7.6
DXOR	Doubleword Logical	4F07	7.6
EQ	Equal Comparison	2A	7.7
GE	Greater Than or Equal Comparison	2C	7.7
LGE	Logical Greater Than or Equal Comparison	3A	7.7
GT	Greater Than Comparison	2D	7.7
LGT	Logical Greater Than Comparison	3B	7.7
LE	Less Than or Equal Comparison	29	7.7
LLE	Logical Less Than or Equal Comparison	39	7.7
LT	Less Than Comparison	28	7.7
LLT	Logical Less Than Comparison	38	7.7
NE	Not Equal Comparison	2B	7.7
DEQ	Doubleword Equal Comparison	4F0A	7.7
DGE	Doubleword Greater Than or Equal Comparison	4F0C	7.7
DGT	Doubleword Greater Than Comparison	4F0D	7.7
DLE	Doubleword Less Than or Equal Comparison	4F09	7.7
DLT	Doubleword Less Than Comparison	4F08	7.7
DNE	Doubleword Not Equal Comparison	4F0B	7.7
FEQ	Floating Point Equal Comparison	4F2A	7.7
FGE	Floating Point Greater Than or Equal Comparison	4F2C	7.7
FGT	Floating Point Greater Than Comparison	4F2D	7.7
FLE	Floating Point Less Than or Equal Comparison	4F29	7.7
FLT	Floating Point Less Than Comparison	4F28	7.7
FNE	Floating Point Not Equal Comparison	4F2B	7.7
SLC	Shift Left Circular	33	7.8
SLL	Shift Left Logical	30	7.8
SRA	Shift Right Arithmetic	31	7.8
SRL	Shift Right Logical	32	7.8
DSLC	Doubleword Shift Left Circular	4F13	7.8
DSLL	Doubleword Shift Left Logical	4F10	7.8
DSRA	Doubleword Shift Right Arithmetic	4F11	7.8
DSRL	Doubleword Shift Right Logical	4F12	7.8
Ln	Load 4 Bit Literal	7n	7.9
LBL	Load Byte Literal	41	7.9
LWL	Load Word Literal	40	7.9
LDL	Load Doubleword Literal	42	7.9
LTL	Load Tripleword Literal	43	7.9
FILL	Fill Stack with Literal	59	7.9
ESW	Enter Configuration Switches	05	7.9

B.3 INDEX TO INSTRUCTIONS - (Cont'd)

An index to instructions is provided.

		Op Code	Topic
STACK OPERATE			
DBL1	Convert TOS to Doubleword	08	7.10
DBL2	Convert TOS2 to Doubleword	4F16	7.10
SNGL	Convert Doubleword to Single Word	37	7.10
DDUP	Duplicate Doubleword on Top of Stack	3F	7.10
DUP	Duplicate Top of Stack	1F	7.10
XCH	Exchange Top of Stack Words	2F	7.10
FLOT	Float an Integer	4C	7.10
FIX	Fix a Floating Point Number	4F17	7.10
GFD	Generate Field Descriptor	2E	7.11
XB1	Convert Index for BIT(1) Arrays	0D	7.12
XB2	Convert Index for BIT(2) Arrays	0E	7.12
XB4	Convert Index for BIT(4) Arrays	0F	7.12
BRANCH INSTRUCTIONS			
BRB	Branch Backward	46	8.2
BRA	Branch	47	8.2
BTOS	Branch Through Top of Stack	15	8.2
DBB	Decrement TOS and Branch Backward	16	8.2
BEQZ	Branch If TOS Equal to Zero	1A	8.2
BGEZ	Branch If TOS Greater Than or Equal to Zero	1C	8.2
DBL	Decrement TOS and Branch Long	17	8.2
BGTZ	Branch If TOS Greater Than Zero	1D	8.2
BLEZ	Branch If TOS Less Than Or Equal to Zero	19	8.2
BLTZ	Branch If TOS Less Than Zero	18	8.2
BNEZ	Branch If TOS Not Equal to Zero	1B	8.2
BRF	Branch False	13	8.2
BRT	Branch True	12	8.2
CASE	CASE Branch	14	8.3
DIB	DO Loop Initialize and Branch	48	8.4
DSBB	DO Loop Step, Branch Backward	4A	8.5
DSBL	DO Loop Step, Branch Long	4B	8.5
CONTROL INSTRUCTIONS			
BENT	Begin Block Entry	53	9.1
BXIT	Begin Block Exit	58	9.1
MARK	Mark Stack for Procedure Call	50	9.2
CALL	Procedure Call	52	9.3
EXIT	Procedure Block Exit	54	9.4
IXIT	Interrupt Procedure Call	55	9.5
RESM	Resume Task in Another Stack	56	9.6

CONTROL INSTRUCTIONS - (Cont'd)		Op Code	Topic
WAIT	Wait for an Interrupt	5C	9.7
SUPV	Supervisor Call	09	9.8
LADR	Load Address	06	9.9
GOTO	GOTO	57	9.10
SSP	Set Stack Pointer	5A	9.11
SSPI	Set Stack Pointer Indirect	5B	9.11
SSR	Stuff Stack Registers	5F	9.11
POP	Pop TOS	0B	9.11
NOP	No Operation	01	9.11
PNOP	Privileged No Operation	02	9.11
TCAR	Test Carry	04	9.11
TOVF	Test Overflow	03	9.11
TRAP	Trap	00	9.11
XIM	Exchange Interrupt Mask	0A	9.11
MICR	Initiate Microprogrammed Procedure	OC	9.12
STRING INSTRUCTIONS			
MOV	Move String Within Stack	4F30	10.2
MVP	Move String from Procedure to Stack	4F31	10.2
MVA	Move String Absolute	4F38	10.2
SLT	String Compare Less Than	4F32	10.2
SLE	String Compare Less Than or Equal	4F33	10.2
SEQ	String Compare Less Than or Equal	4F34	10.2
SNE	String Compare Not Equal	4F35	10.2
SGE	String Compare Greater Than or Equal	4F36	10.2
SGT	String Compare Greater Than	4F37	10.2

B Index and Tables

B.4 INSTRUCTION OP CODE TABLE

A table of instruction Op Codes is provided.

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X	TRAP	NOP	PNOP	TOVF	TCAR	ESW	LADR		DBL1	SUPV	XIM	POP	MICR	XB1	XB2	XB4
01	NEG	NOT	BRT	BRF	CASE	BTOS	DBB	DBL	BLTZ	BLEZ	BEQZ	BNEZ	BGEZ	BGTZ	ABS	DUP
2X	ADD	SUB	MUL	DIV	MOD	AND	OR	XOR	LT	LE	EQ	NE	GE	GT	GFD	XCH
3X	SLL	SRA	SRL	SLC	MAX	MIN	MULD	SNGL	LLT	LLE	LGE	LGT	DNEG	DNOT	DABS	DDUP
4X	LWL	LBL	LDL	LTL	SGN	DSGN	BRB	BRA	DIB		DSBB	DSBL	FLOT			*
5X	MARK		CALL	BENT	EXIT	IXIT	RESM	GOTO	BXIT	FILL	SSP	SSPI	WAIT			SSR
6X																
7X	L0	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15
8X	←	←	←	STD	→	→	→	→	←	←	←	←	STF	→	→	→
9X	←	←	←	SWAP	→	→	→	→	←	←	←	←	SST	→	→	→
AX	←	←	←	AWM	→	→	→	→	←	←	←	←	STWN	→	→	→
BX	←	←	←	STW	→	→	→	→	←	←	←	←	STB	→	→	→
CX	←	←	←	LD	→	→	→	→								
DX	←	←	←	LF	→	→	→	→	←	←	←	←	LTW	→	→	→
EX	←	←	←	AW	→	→	→	→	←	←	←	←	SW	→	→	→
FX	←	←	←	LW	→	→	→	→	←	←	←	←	LB	→	→	→

*	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X	DADD	DSUB	DMUL	DDIV	DMOD	DAND	DOR	DXOR	DLT	DLE	DEQ	DNE	DGE	DGT	DMAX	DMIN
1X	DSLL	DSRA	DSRL	DSLC	DIVD	MODD	DBL2	FIX	FABS	FSGN						
2X	FADD	FSUB	FMUL	FDIV					FLT	FLE	FEQ	FNE	FGE	FGT	FMAX	FMIN
3X	MOV	MVP	SLT	SLE	SEQ	SNE	SGE	SGT	MVA							