Burroughs

B 5900 System

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. September 1981 5011034

Burroughs

B 5900 System

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. September 1981 5011034

Burroughs believes that the information described in this publication is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, can be accepted for any consequences arising out of the use of this material, including loss of profit, indirect, special, or consequential damages. There are no warranties which extend beyond the program specification.

The Customer should exercise care to assure that use of the information in this publication will be in full compliance with laws, rules and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change. Revisions may be issued from time to time to advise of changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to
TIO West Documentation, Burroughs Corporation,
1300 John Reed Court, City of Industry
California 91745
U.S.A.

LIST OF EFFECTIVE PAGES

Page	Issue
Title	Original
ii	Original
iii	Original
iv	Blank
v thru xv	Original
xvi	Blank
xvii	Original
xviii	Blank
1-1 thru 1-14	Original
2-1 thru 2-16	Original
3-1 thru 3-17	Original
3-18	Blank
4-1 thru 4-77	Original
4-78	Blank
5-1 thru 5-27	Original
5-28	Blank
6-1 thru 6-35	Original
6-36	Blank
A-1 thru A-9	Original
A-10	Blank
B-1 thru B-2	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Cont)

LIST OF TABLES

INTRODUCTION

The B 5900 is designed to extend the family of Burroughs large systems computers downward into the medium systems cost and performance range. The B 5900 reduces the size and environmental requirements for a large computer system. In comparison with the B 6800, the B 5900 requires 76 percent less Floorspace, and 80 percent less power consumption and air conditioning. The B 5900 utilizes large scale integration devices throughout and subsequently requires only a single cabinet to house the entire system.

The B 5900 system utilizes the same dynamic storage allocation concept that has proven successful in Burroughs Large Systems previously. This concept utilizes a Descriptor method of segmentation which allows variable length segments of data to be used. This method is more efficient than "fixed-size" paging concepts.

The B 5900 system contains the capability to be interfaced with, and to operate from GLOBAL™ memory applications.

The sections of this manual are organized in the following manner:

SECTION 1, SYSTEM DESCRIPTION, discusses the design of the B 5900 Central Power Cabinet and the independent sub-systems and architecture.

SECTION 2, DATA REPRESENTATION, discusses data representation and word formats.

SECTION 3, STACK CONCEPT, discusses stack concept as implemented in large systems and reverse polish notation.

SECTION 4, B 5900 OPERATOR SET, details the implementation of the large system operator set as it applies to the B 5900.

SECTION 5, SYSTEM CONCEPT, discusses the functional aspects of the B 5900 system.

SECTION 6, INPUT/OUTPUT DATA COMMUNICATION SUBSYSTEM, discusses the IODC subsystem interfaces.

APPENDICES, Operator set (alphabetical and numerical representation), and Data representation

The B 5900 operation is performed by a specially designed microprogram. Since the microprogram encompasses a large part of the functional explanation, discussion will make reference to microprogramming. Under no condition will an attempt to describe microprogramming be made. Microprogram references will be included only to clarify or explain machine operation that can not be adequately described from a hardware point of view. If a part of the system can be discussed in both hardware and microprogram terms, this manual will use the hardware explanation.

GLOBAL is a trademark of Burroughs Corporation

SECTION 1
SYSTEM DESCRIPTION

GENERAL

This manual describes the B 5900 Information Processing system. The B 5900 is designed to open an entry level position in Burrough's large systems computers. Object code compatible with B 6000/B 7000 series systems, the B 5900 utilizes advanced architecture and employs medium and large scale integration (MSI AND LSI) logic devices.

This manual explains how the B 5900 Information Processing System achieves flexibility and efficiency through a comprehensive system approach to problem solving, without considering the areas of computer logic or circuit design. The B 5900 is a compiler oriented system, designed to accept the high level problem-solving language compilers such as ALGOL, COBOL, FORTRAN, and PL/I.

The B 5900 system operates under the control of a Master Control Program (MCP), which automatically handles memory assignments, program segmentation, and subroutine linkages. The use of the MCP eliminates many arduous programming tasks which are likely to produce errors. The compilers operate under the control of the MCP, as do the object-programs that result from the use of the compilers. The programs are debugged and corrected in the source language.

B 5900 SYSTEM HARDWARE ORGANIZATION

This section discusses the Central Processing Cabinet (CPC) of the B 5900 and the various modules located in the cabinet. The central processor, the local memory modules, and the IODC subsystem are housed in this single cabinet.

There are six backplanes in the Central Cabinet. One Central Processor, one Local IC Memory (up to 512 KWords), three I/O Base Modules, and an optional 512 KWord IC Memory. Figure 1-1 shows the basic configuration of the Central Processing Cabinet backplanes and interface boxes. A minimum of two I/O Base Modules are configured in the basic B 5930 system.

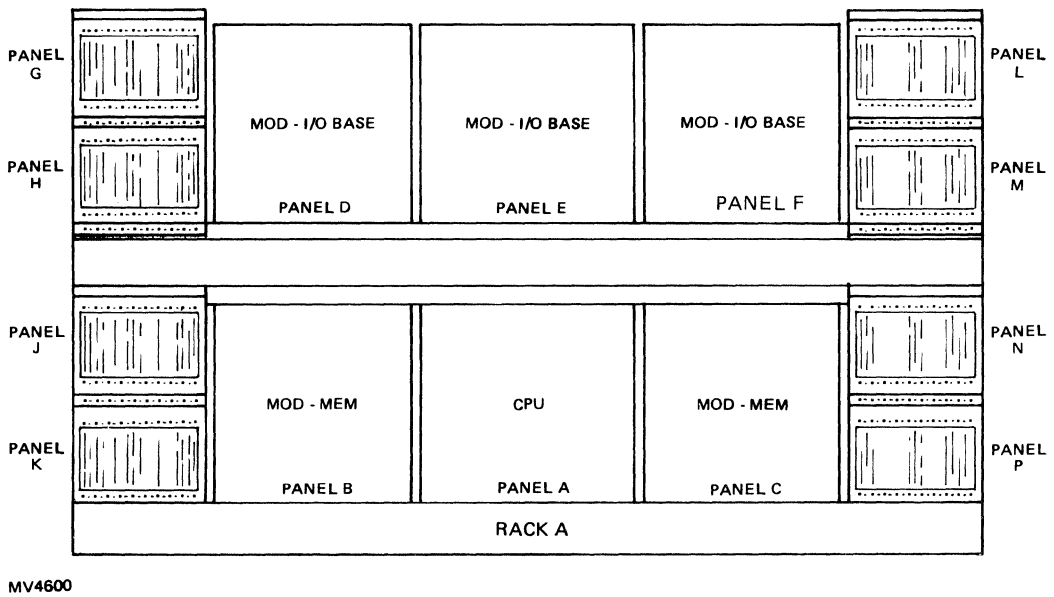


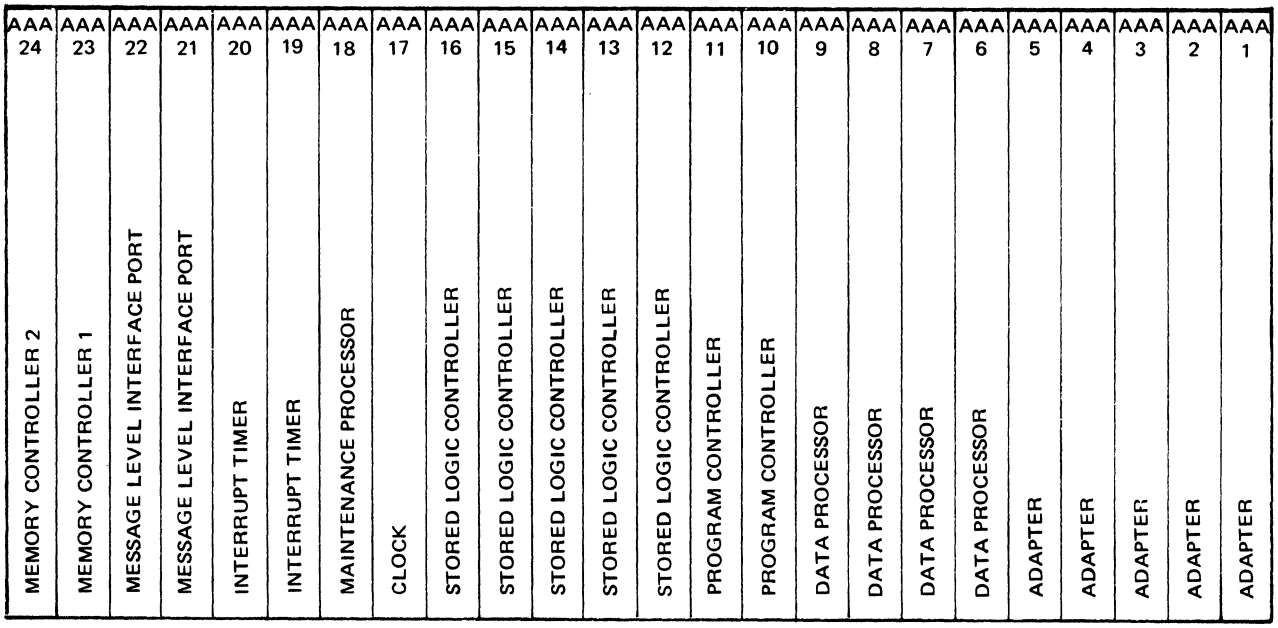
Figure 1-1. B 5900 CPC Backplanes

CENTRAL PROCESSOR

The Central Processor is comprised of 17 cards resident in a 24-slot backplane. The remaining seven slots are for interface adapters to other subsystems. The Processor is designed as a set of individual functional modules: The Stored Logic Controller (SLC; 5 boards), the Data Processor (DP; 4 boards), the Message Level Interface Port (MLIP; 2 boards), the system Clock (1 board), the Program Controller (PC; 2 boards), the Maintenance Processor (MP; 1 board), and the Interrupt/Timer (I/T; 2 boards). The interface adapters are the optional Host Global Memory Interfaces (3 boards), the Memory Controls (2 boards plus 1 located in the memory module), and 2 more slots for Memory Control in case the extended memory option is utilized. The modules of the Central Processor reside in the designated slots as shown in Figure 1-2.

The Central Processor has two main buses. The 52-bit Main Data Bus (M-bus) and the 30-bit Control Bus (C-bus). The M-bus is used to transfer data between the separate processor modules. The C-bus originates in the SLC and carries uni-directional command information to the other modules.

The processor contains the generation circuitry for the system clock pulse. The clock card contains a 20.0 megahertz oscillator. The system clock frequency for the B 5900 system is 5.0 megahertz. This frequency propagates clock pulses every 200 nanoseconds. There are two minor clocks pulses of 100 nanoseconds and 50 nanoseconds.



MV4601

Figure 1-2. Backplane Organization of Processor Modules

Figure 1-3 shows the overall system architecture of the B 5900. The system communicates by way of the MLI I/O interface, a Global Memory interface, and an expansion memory interface. The MLI and maintenance interfaces are provided with the basic system. The additional memory interfaces are available as adapters.

Figure 1-4 depicts the internal architecture of the B 5900 processor. The modules which make up the processor communicate by way of the two above mentioned buses (M-bus and C-bus). The Data Processor also uses the M-bus for certain internal operations. The Bus Control module, located in the Data Processor, responds to program controlled allocation of the M-bus by the SLC, and asynchronous requests for data transfer by the memory modules. Data on the M-bus, in the memory and in the register files, is positive when true. That is, the logic level for Data is positive if logical ONE and negative if logical ZERO.

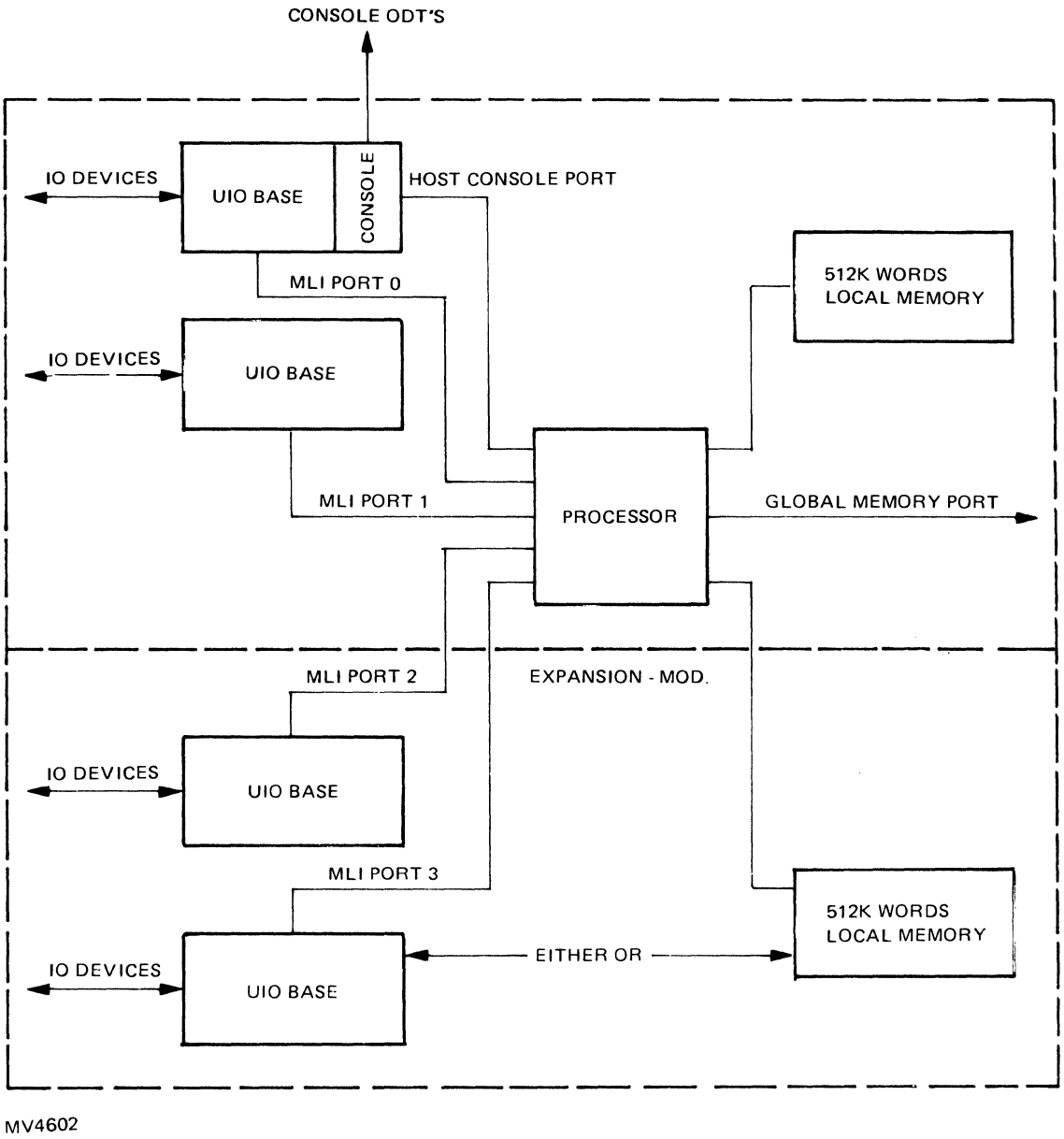


Figure 1-3. B 5900 System Architecture

The C-bus is a 30 wire control bus that transfers commands from the SLC to the other modules in the processor. The interpretation of the command on the C-bus is controlled by a 3-bit field that indicates which module must respond to the command.

The B 5900 has several maintenance features which include the integrated Maintenance Processor which runs at system clock speed, and an external Maintenance Interface Processor located in the operator console. All PROMs and RAMs in the processor are parity checked; all data buses in the system are parity checked on every clock cycle of the machine operation. All registers can be read and set from the Maintenance Processor. The memory subsystem includes single-bit error correction and multiple-bit error detection. Memory error correction occurs in-line, without affecting the system, and the hardware logs memory errors.

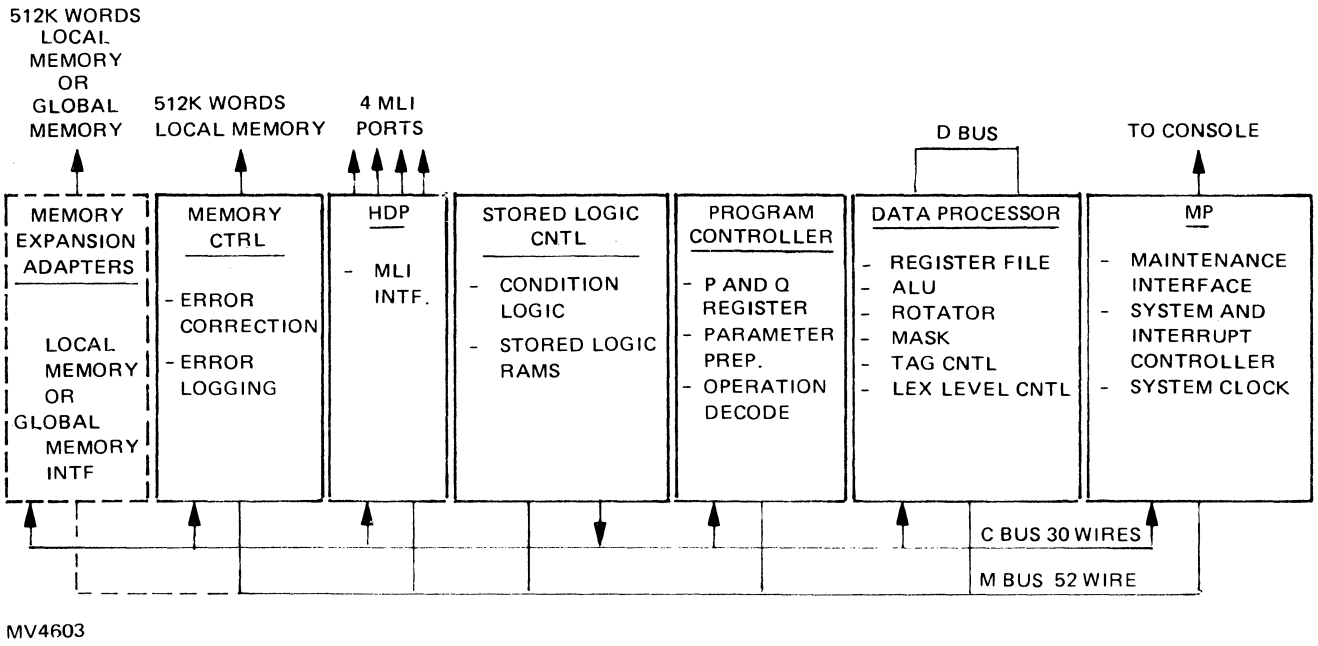


Figure 1-4. B 5900 Processor Internal Architecture

The 200 nanosecond system clock has a 35 nanosecond pulse width. The structure of the processor is such that one data transformation occurs with each clock. All module-to-module communications are relative to the low-to-high transition of the system clock. The system and minor clocks may be stopped or single pulsed independently. The processor clocks may be stopped without affecting the memory clocks so as to retain any valid data in memory.

The I/O function is supported by the SLC. During an I/O burst, no other processing occurs. Because there is only a single interface to memory (the M-bus), simultaneous processing and I/O transfer are prevented.

INPUT/OUTPUT AND PERIPHERALS

The Input/Output Data Communications (IODC) subsystem interfaces peripheral devices with the host B 5900 system. The IODC subsystem can be maintained without regard to the host system architecture.

The basic IODC subsystem configuration is shown in Figure 1-5. The Host is the central processor that controls the activity of the IODC subsystem by supplying commands, granting access for data transmissions, and receiving results. The portion of the CPU that directly controls the subsystem is designated as the Message Level Interface Port (MLIP). The MLIP communicates with the IODC subsystem by way of Message Level Interfaces. The unit at the opposite end of the MLI inside of the I/O base is the Connection Module (Conmod) which is a collective name for a group of units that are responsible for connecting the host to subsystem destinations (Base Control Card, Distribution Card, and in a multiple processor system a Path Selection Module). These destinations are called Data Link Processors (DLP) and are the units that control the activity of peripheral devices and data communication channels. Certain destinations in the IODC base module have DLPs as their peripheral devices and are called Network Support Processors (NSP). The Connection Module is connected to the DLPs by way of Data Link Interface (DLI). The The DLP communicates to peripherals across requisite Peripheral Interfaces (PI).

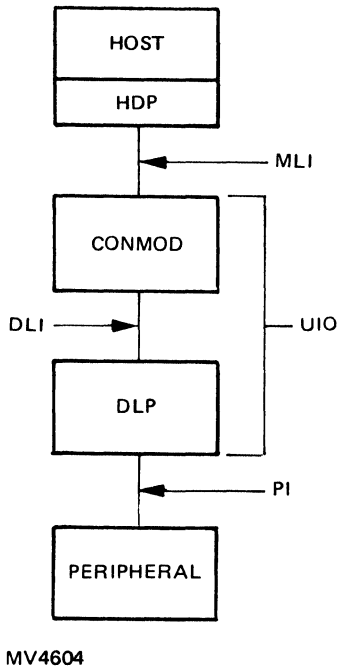


Figure 1-5. Basic IODC Configuration

Figure 1-6 shows an expanded view of the Connection Module within the IODC subsystem. The basic unit within the Conmod is the Distribution Card (DC). This unit follows the MLI Protocol to connect a host to a DLP and converts the asynchronous MLI to the synchronous DLI. I/O units reside within the Base Module (BMOD). (See Figures 1-7 and 1-8.) A maximum of 8 DLP are capable of existing in a Base Module. The maximum number of Base Modules that can exist within the subsystem is dependent on the host capabilities and requirements. A maximum of 6 Distribution Cards can also be present within a single Base Module. Each DC is connected to a separate MLIP; either within the same host system or within multiple hosts.

If more than one DC is present within a Base Module, the Path Selection Module is required. The PSM controls access from the DC to the DLP and assures that a DLP communicates to the proper host system.

The Base Control Card (BCC) is an optional I/O unit that provides a mechanism for identifying the base modules to the hosts. The BCC provides masking that disables one or more MLI, assigns DLPs to particular hosts, and disables or enables the maintenance interface to each unit within the base module.

The Base Control Card, the Path Selection Module, and all Distribution Cards present within the base module are connected by way of the Connection Module Interface (CMI). If the Connection Module consists of a single Distribution Card without the Base Control Card, this interface does not exist.

The Line Expansion Module (LEM) is a collective name for several I/O units that provide fan-out capability on the MLI. The MLI can not be daisy-chained. A single unit must reside at each end of the point-to-point interface. Therefore, without the LEM, a single MLIP can communicate to a single DC and, to a maximum of 8 DLPs. The LEM provides interface capability from a single MLIP to a maximum of 8 DCs (64 DLPs) using up to 9 MLIs and an exchange mechanism.

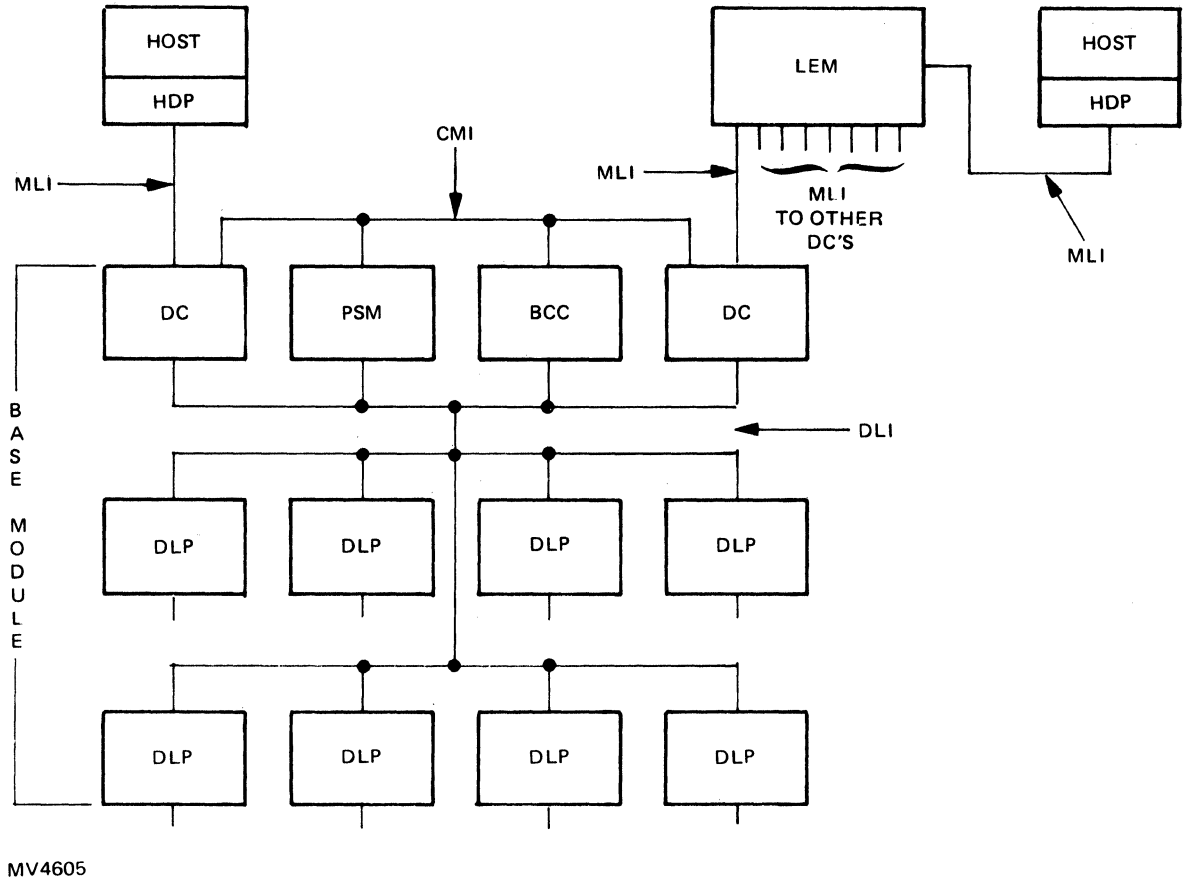


Figure 1-6. Expanded IODC Configuration

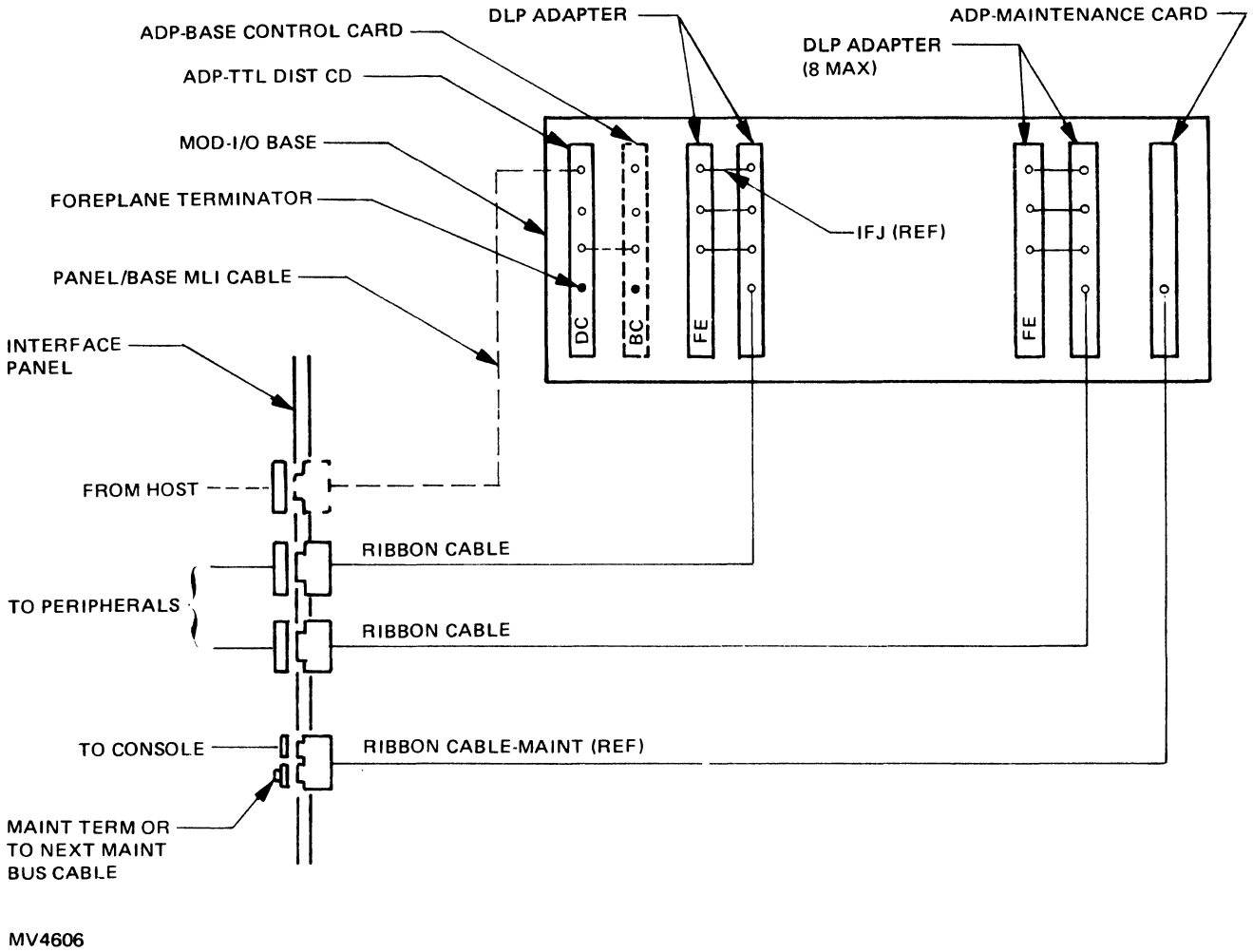


Figure 1-7. Single I/O Base Configuration (Single MLIP Interface)

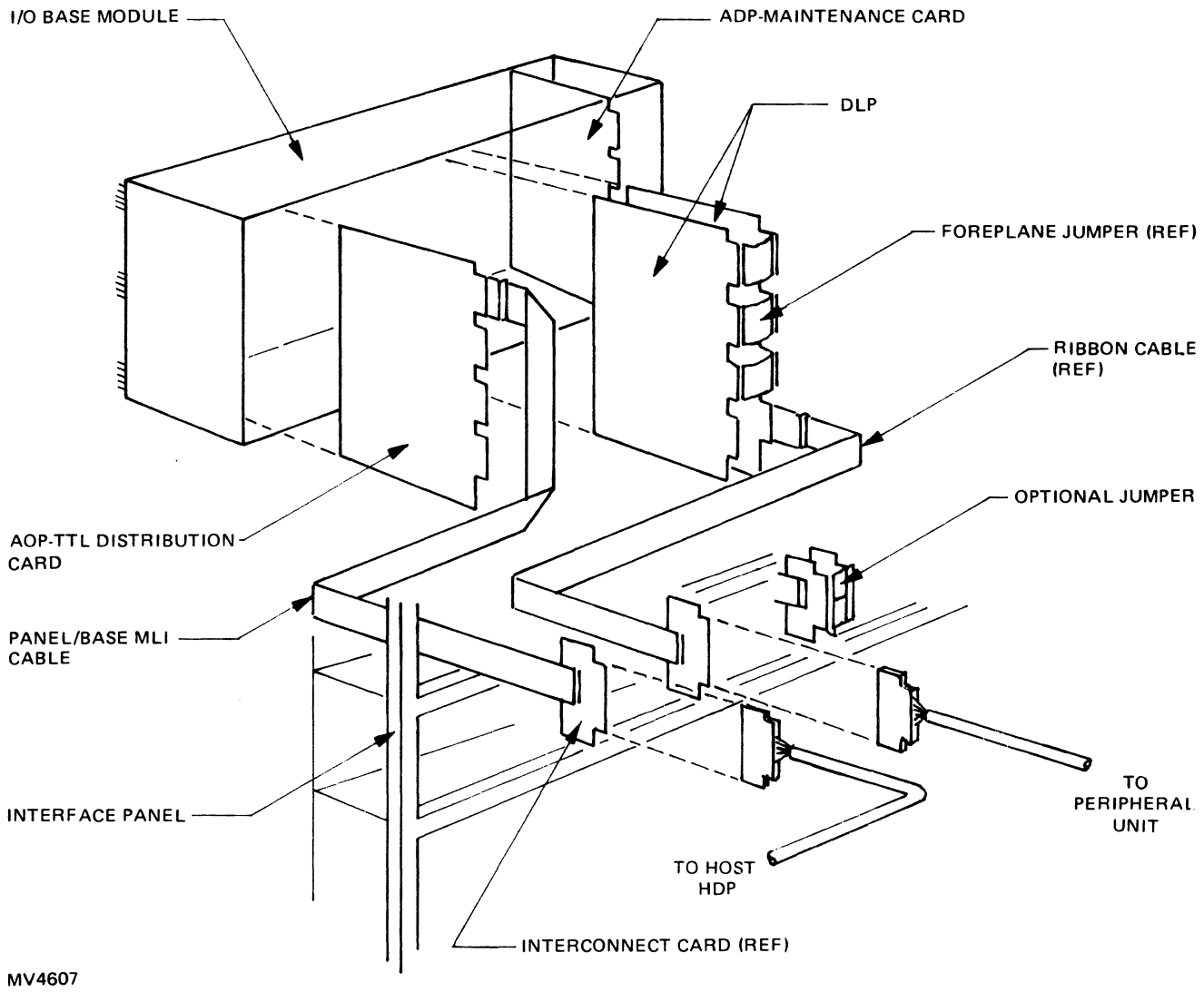


Figure 1-8. I/O Base Module

The IODC provides a standard maintenance program. The special hardware used for this purpose is shown in Figure 1-9. The primary unit is the Maintenance Card (MC). This unit generates the master clocks for the base module within which it is housed. It also contains most of the common maintenance functions used by IODC. The MC communicates to all other units within the same base module by way of the Maintenance Interface. The Maintenance Card is connected to the Maintenance Interface Processor (MIP) by way of the Test Bus. Up to 63 MCs are connected on this bus. The MIP drives diagnostic routines across the Test Bus to the Maintenance Cards. Figure 1-9 shows how the front-end MLI and back-end Test Bus interface into I/O base modules within the overall IODC subsystem.

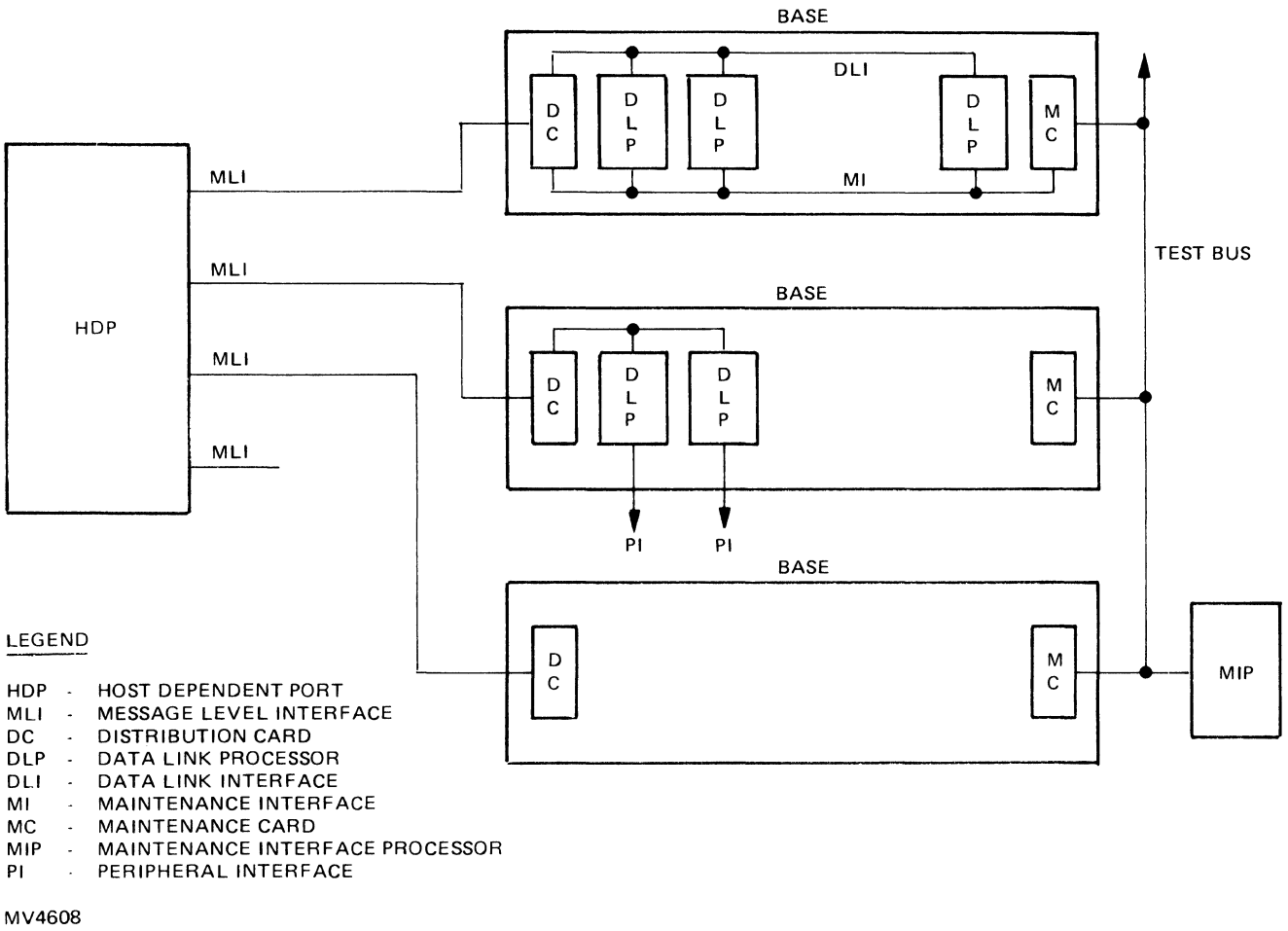


Figure 1-9. B 5900 IODC MLI and Test Bus Interfaces

Figure 1-10 shows the I/O and Console subsystems schematically. Details of the configuration are specified and described below.

B 5900 Operator Console V

The Operator Console V provides a central location for all system operating controls. The locating of normal operating controls at a single central location is efficient, and provides a logical place for the system operations staff to function.

The Console V is integrated into an I/O backplane, providing the means for the user to access the console for normal operator functions or the mini-disk for maintenance purposes.

The operator set of the B 5900 is microprogrammed to provide compatibility with the B 6000/B 7000 operator set. Any program written on a B 5900 system with Burroughs large systems compilers and MCPs will run on any other system in the large systems product line.

The microprogram is loaded by way of floppy disk every time power is applied. Special microprograms can be loaded by the Field Engineer to perform maintenance and diagnostic routines.

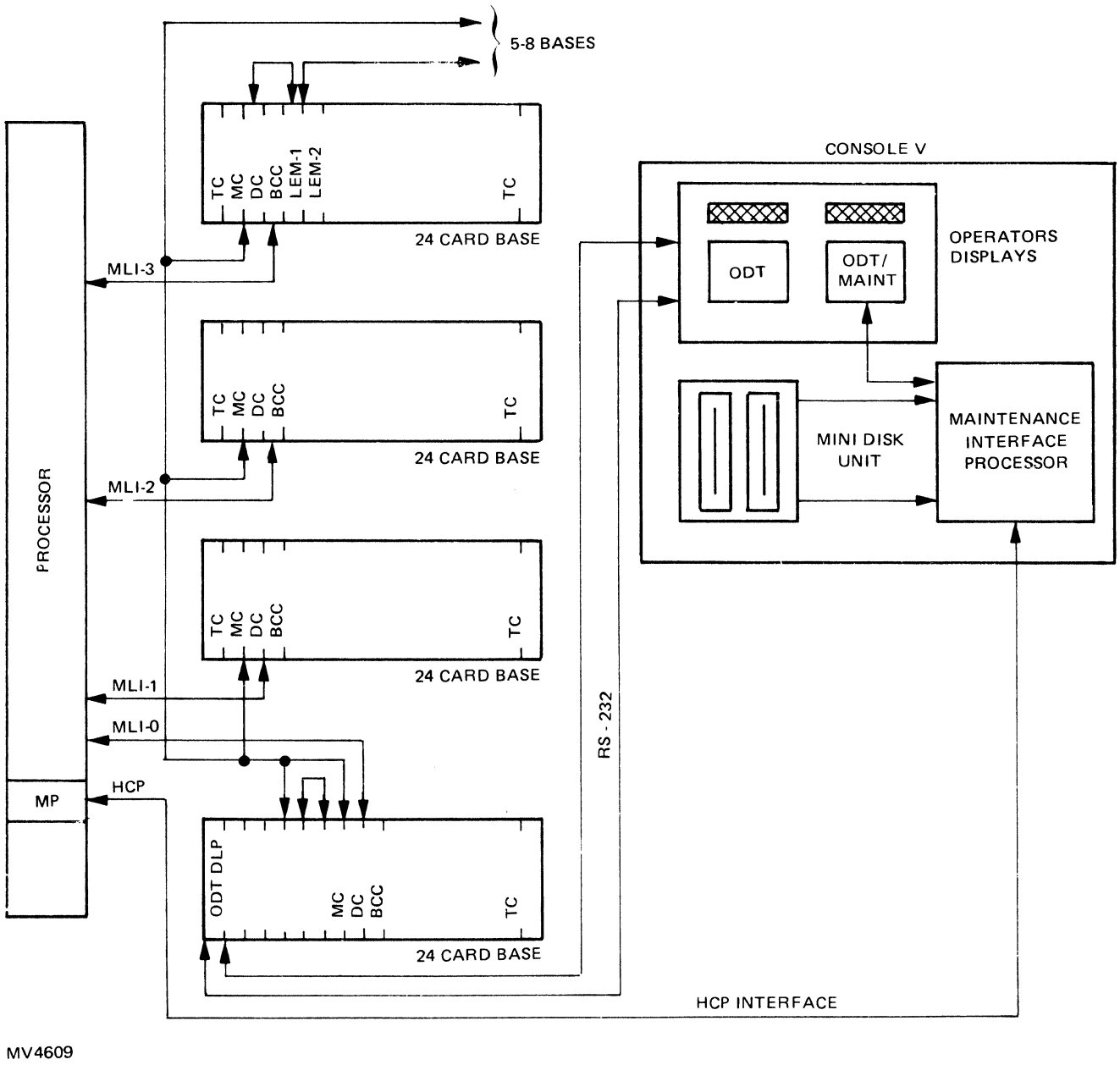


Figure 1-10. B 5900 I/O and Console Subsystem Schematic

Figure 1-11 shows the B 5900 Operator Display Console. The operating system functions in two modes: Normal(ODT) mode and Maintenance mode. The operator display console has two stations. The left display is for ODT operation only and the right display can be used for maintenance or ODT operation. Both displays have a locking device that disables keyboard function but does not disable the operator control panel. Figure 1-12 is a representation of the B 5900 operator keyboard. The keyboard indicator LEDs are located above and to the left of the Control, Specify, Local, Receive, and Transmit buttons. The labels on the indicators are valid for Normal (ODT) mode. In Maintenance mode LTAI indicates activity on the flexible disk system. ENQ indicates flexible disk error. The FORMS, LOCAL, RCV, and XMT are not used and any indication may be due to ODT problems.

The keyboard is used by a system operator to enter commands and data to the operating system. The operators console and keyboard are commonly referred to as an Operators Display Terminal (ODT).

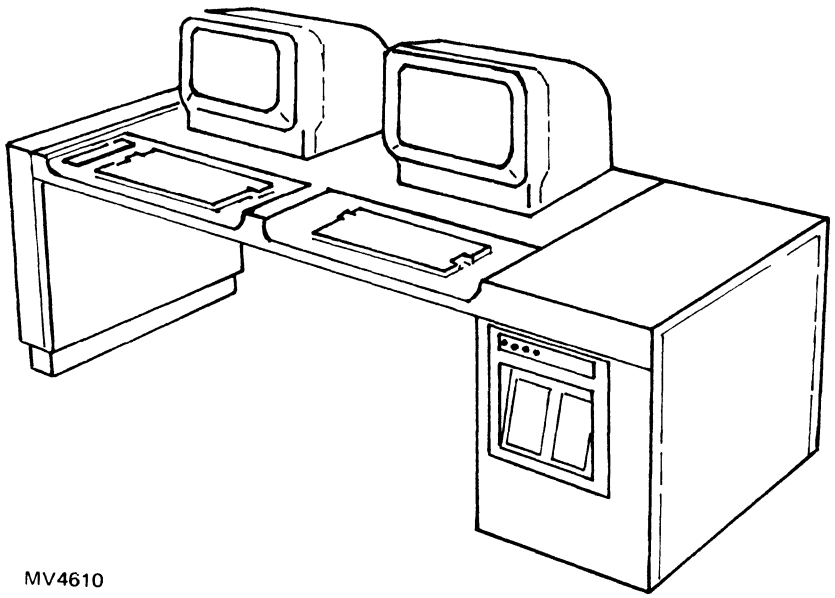
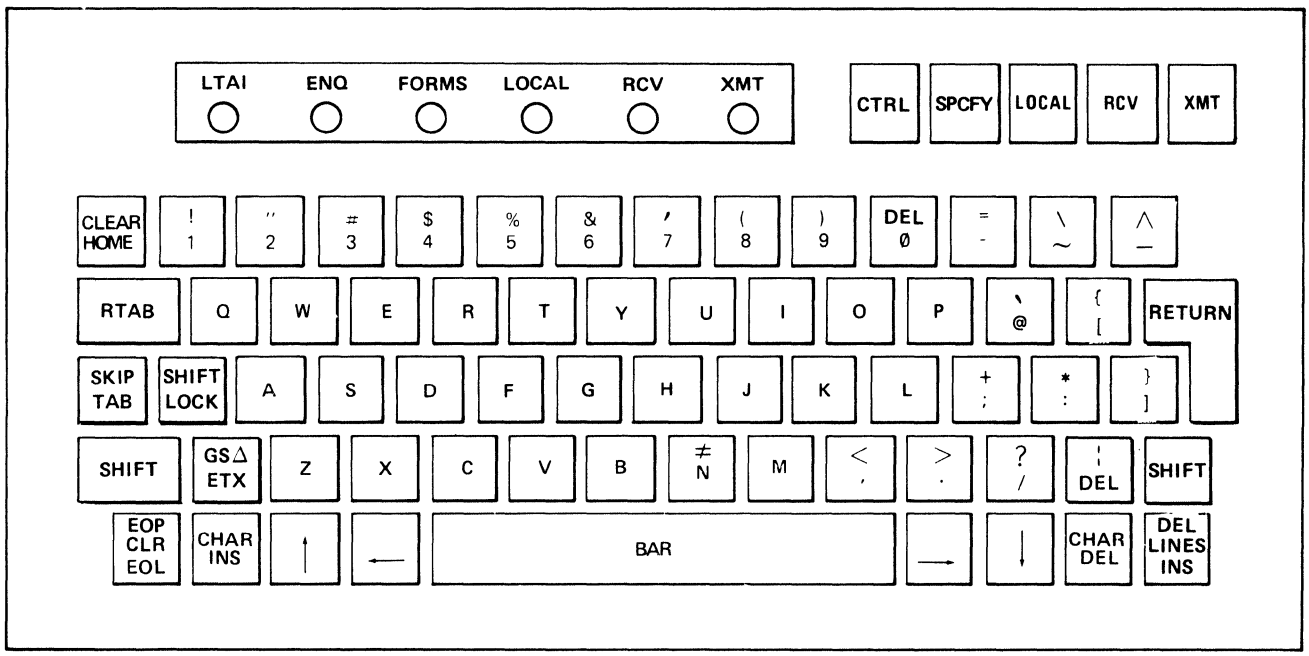


Figure 1-11. B 5900 Operator Display Console



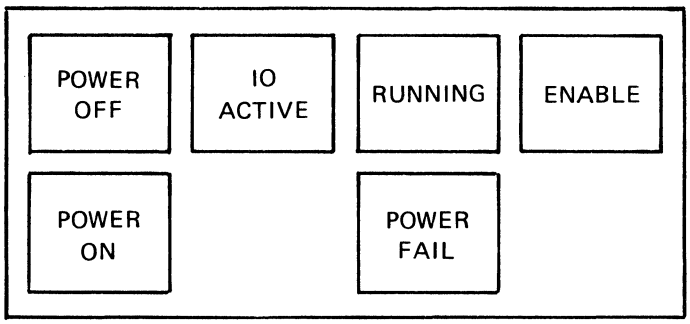
MV4611

Figure 1-12. B 5900 Operator Keyboard

When the operator needs to communicate with the operating system, the keyboard is used to write data which is displayed on the screen. The screen is capable of displaying 1920 characters, which are arranged in a matrix that consists of 24 rows of characters. Each row of characters contains 80 character positions. A cursor moves from left to right, and from top to bottom on the screen. The display screen has automatic line-feed, and carriage-return features so that the operator is not required to control these functions. When the operator writes data on the screen, the last character written must be the End-of-Text special character (ETX) which moves the cursor to the first letter of the text to be transmitted, and indicates to the system where the communication is to stop.

The controls for the B 5900 system consist of six indicator/switch pushbuttons shown in Figure 1-13. The function of each control is as follows:

- The ENABLE pushbutton allows the use of the POWER ON and POWER OFF pushbuttons. If the ENABLE pushbutton is not depressed then the other two pushbuttons listed are inoperative, and have no effect on system operation. If the ENABLE pushbutton is depressed then the other two pushbuttons are operative. The purpose of the ENABLE pushbutton is to prevent accidental system operation by inadvertent depression of one of the control buttons.
- The POWER OFF pushbutton is used to remove source power from the Central Processing Cabinet and the Console. The Console may be strapped so that it remains on when the POWER OFF is depressed.
- The POWER ON pushbutton is used to apply source power to the Central Processing Cabinet and to the Console. The POWER ON pushbutton does not apply power to peripherals or other cabinets.
- The RUNNING indicator is illuminated when the system is operating normally. If power is on and the light is not illuminated a looping condition is indicated.
- The IO ACTIVE indicator is connected to the Message Level Interface and illuminates when there is activity on these interfaces.
- The POWER FAIL indicator is illuminated when a power failure occurs in the main cabinet.



MV4612

Figure 1-13. Console Operating Controls

B 5900 Peripheral Devices

The basic B 5930 system includes the following components:

- Central Processing Cabinet
- Console V. operators' console with dual displays, two ICMD mini-disks, and the Maintenance Interface Processor.
- 2 I/O base modules with space for up to 16 DLPs.
- 1 Line Printer DLP
- 1 Magnetic Tape DLP
- 2 Disk Pack DLPs
- 1 ODT DLP
- 1 Line Support Processor
- 1 Quad Line Adapter supporting four communication lines and four Electrical Interfaces

Additional peripheral devices available for the B 5900 include the following:

Card Reader and DLP
Card Punch and DLP
PE Magnetic Tape and DLP
NRZ Magnetic Tape and DLP
GCR/PE Magnetic Tape and DLP
5N Disk and DLP
Line Support Processor
Network Support Processor

Each B 5900 basic system is equipped with two I/O base modules, and is expandable to four base modules with additional DLP configuration. (See Figure 1-14.) Each I/O base module provides space for up to eight DLPs. Since a basic system provides expansion to four I/O bases, each capable of housing eight DLPs, the maximum single processor system provides up to 32 DLP positions. Each peripheral oriented DLP requires a single DLP address position in an I/O base. The basic B 5900 requires five DLP address positions to house the basic peripheral oriented DLPs in that configuration (that is, ODT DLP, Line Printer DLP, Magnetic Tape DLP, and two Disk Pack DLPs. See Figure 1-14).

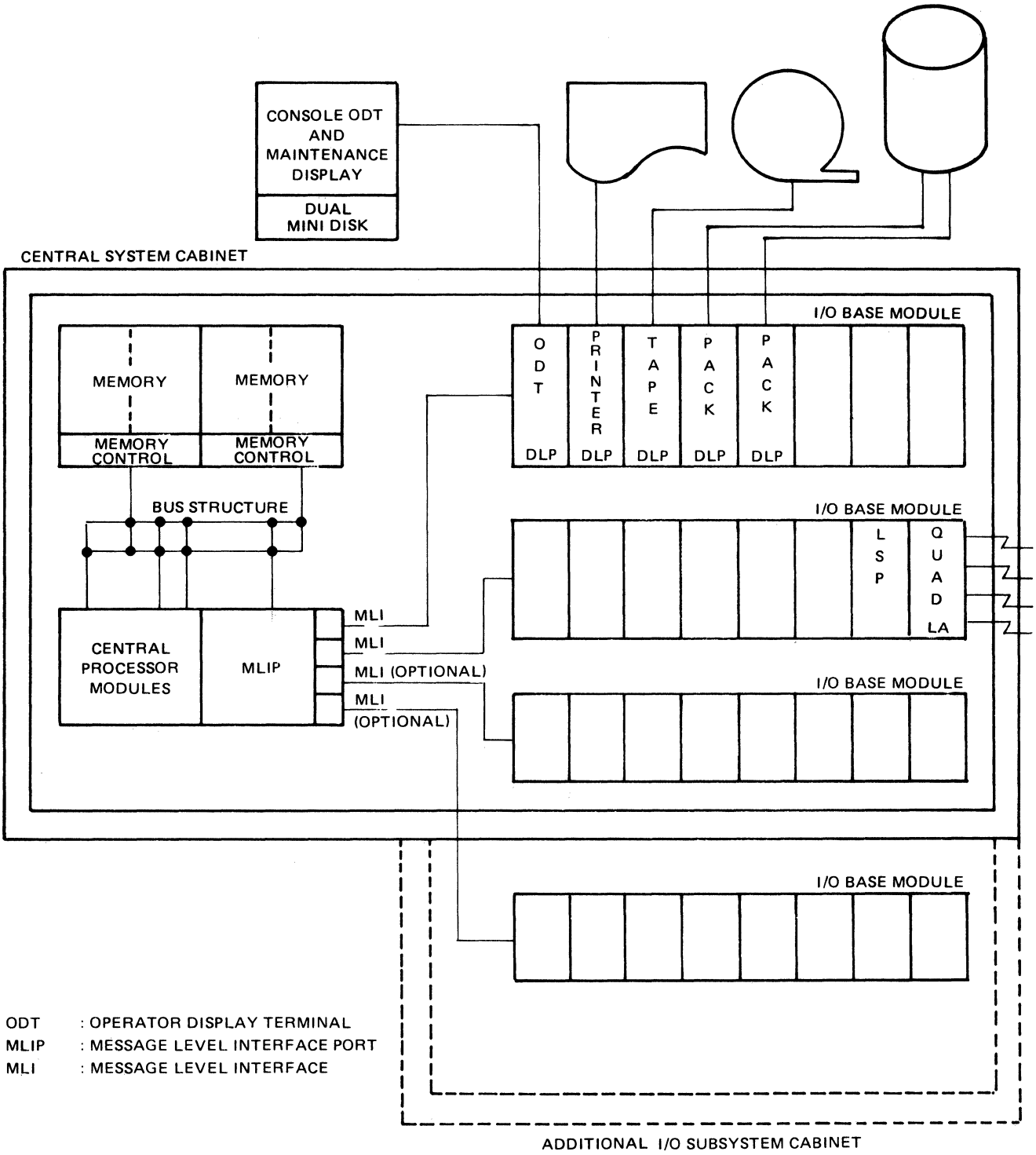


Figure 1-14. Block Diagram of Basic B 5930 System

SECTION 2
DATA REPRESENTATION

GENERAL

All data in the B 5900 system is in binary form. The basic unit of data is the word (see Figure 2-1), which consists of 52 consecutive binary bits. All words of data in the B 5900 system have three distinct parts: a parity bit, three tag bits, and 48 bits of information field. The 52 bits in a word are numbered for identification.

Bit number 51 (the most significant bit in a word) is the parity bit. The parity bit is used to give the word odd parity. If the number of binary ONEs present in the tag and information fields is an even number then the parity bit is a binary ONE value. If the number of binary ONEs present in the tag and information fields is an odd number then the parity bit is a binary ZERO value. The B 5900 system uses the parity bit to monitor the quality of data in a word. Logic circuits count the number of bits in a word and compare the count against the parity bit state. If the result of the comparison is not equal, then the B 5900 system recognizes that a parity error has occurred. The process of parity checking is an automatic function of the system. The parity bit for a word is not directly available to the user because it is used only when words are transferred from one module to another. Data that is internal to a module has already been tested for parity.

Bits 50, 49, and 48 are the tag field. The tag field is used to identify the type of interpretation that is to be applied to the data that is present in the information field of the word. There are eight different values that may be present in tag field, and each value specifies a different interpretation to be used. The tag field values are defined in the following paragraphs.

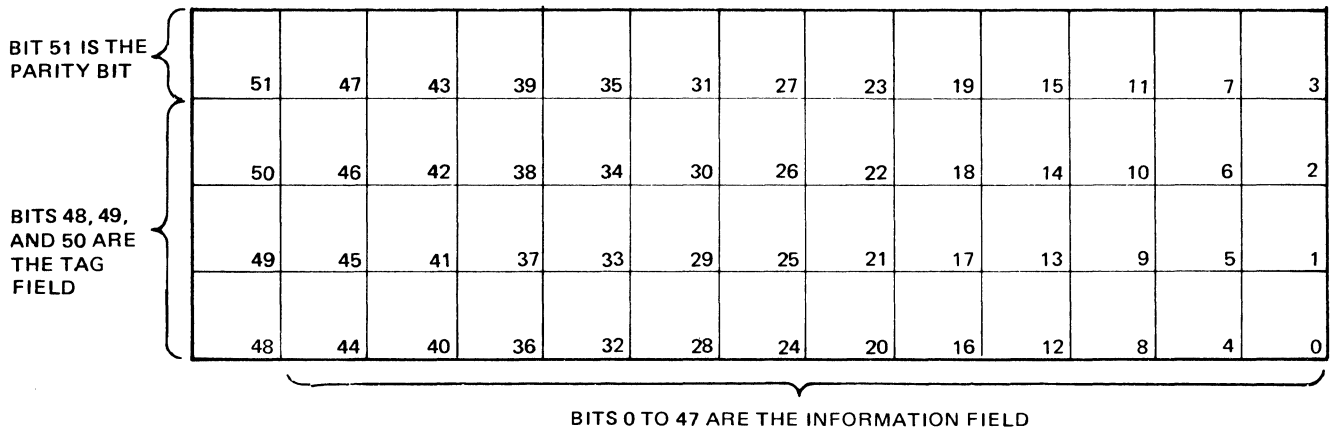


Figure 2-1. B 5900 Word Structure

INTERNAL CHARACTER CODES

The B 5900 uses several different character codes (See Figure 2-2). The primary internal character code is Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC is an 8-bit alphanumeric code containing four zone bits, followed by four numeric bits.

The primary character code used for Data Communications Subsystems is the American Standard Code for Information Interchange (ASCII). ASCII may be either a 6-bit, 7-bit, or 8-bit alphanumeric code. Within the B 5900 system, numeric data may be compacted by deleting the zone bits and retaining the numeric portion of the character. When data is compacted it is said to be packed.

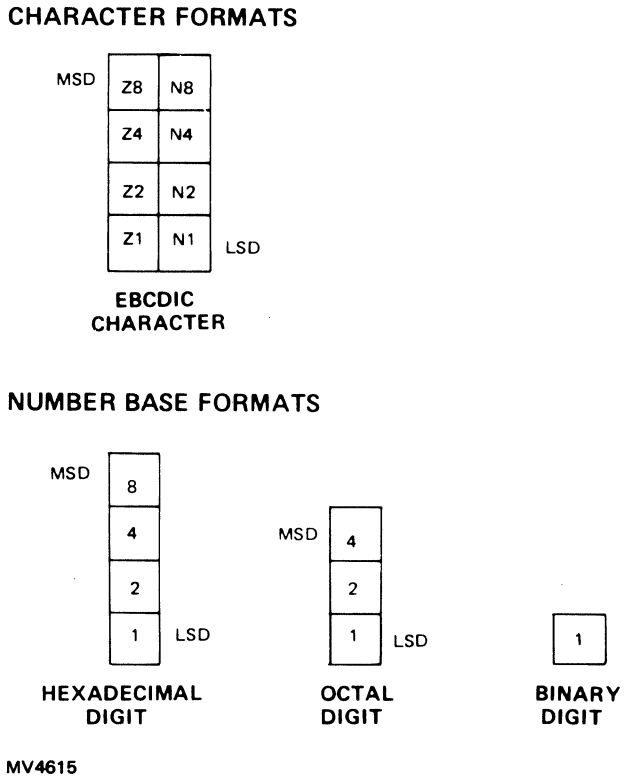


Figure 2-2. Character and Digit Formats

NUMBER BASES

Number bases used in the B 5900 system are base 10 (decimal), base 16 (hexadecimal), base 2 (binary), and base 8 (octal) (see Figure 2-2). Because the system utilizes these various number bases in performing its functions, it is necessary that the user of the system be familiar with the number bases, and know how to convert a value from one number base to any of the others. A brief discussion of the number systems used is given below.

The decimal numbering system is based on the numeric digits 0 through 9, and on the powers of 10. Similarly, the binary numbering system is based on the numeric digits 0 and 1, and on the powers of 2. In this case, a decimal digit may have any value from 0 through 9, and a binary digit may have a value of 0 or 1.

The octal numbering system is based on the numeric digits 0 through 7, and on the powers of 8. An octal digit may have any value from 0 through 7. Furthermore, 2 raised to the third power is 8, the base of the octal numbering system. Because the octal numbering base is a multiple of the binary numbering base, octal to binary and binary to octal conversions are easily accomplished.

The hexadecimal numbering system is based on the numeric digits 0 through 9, and alpha characters A through F; where A equals decimal 10, B equals decimal 11, and so forth to F, which equals decimal 15. Hexadecimal numbering is also based on the powers of 16. 2 raised to the fourth power is 16, the base of the hexadecimal numbering system. Because the hexadecimal numbering base is a multiple of the binary numbering base, a hexadecimal number can be converted to a binary number conveniently.

A B 5900 word contains 48 bits in the value field (see Figure 2-3). These 48 bits can be converted into hexadecimal, octal, or EBCDIC values by arrangements of the 48 bits in the proper order. A Hex digit is equivalent to four binary digits, that is, binary 1111 is equal to hexadecimal F. Since a hexadecimal digit contains four binary digits, the value field of a B 5900 word contains 12 complete hexadecimal digits (48/4=12). The same value field can also be considered to contain 16 octal digits (48/3=16), or 6 EBCDIC characters (48/8=6).

From the preceding discussion it is clear that the choice of 48 bits for the value field of a B 5900 word was not a random choice, but was chosen because the number is a multiple of the common character codes and number bases used in the B 5900 system.

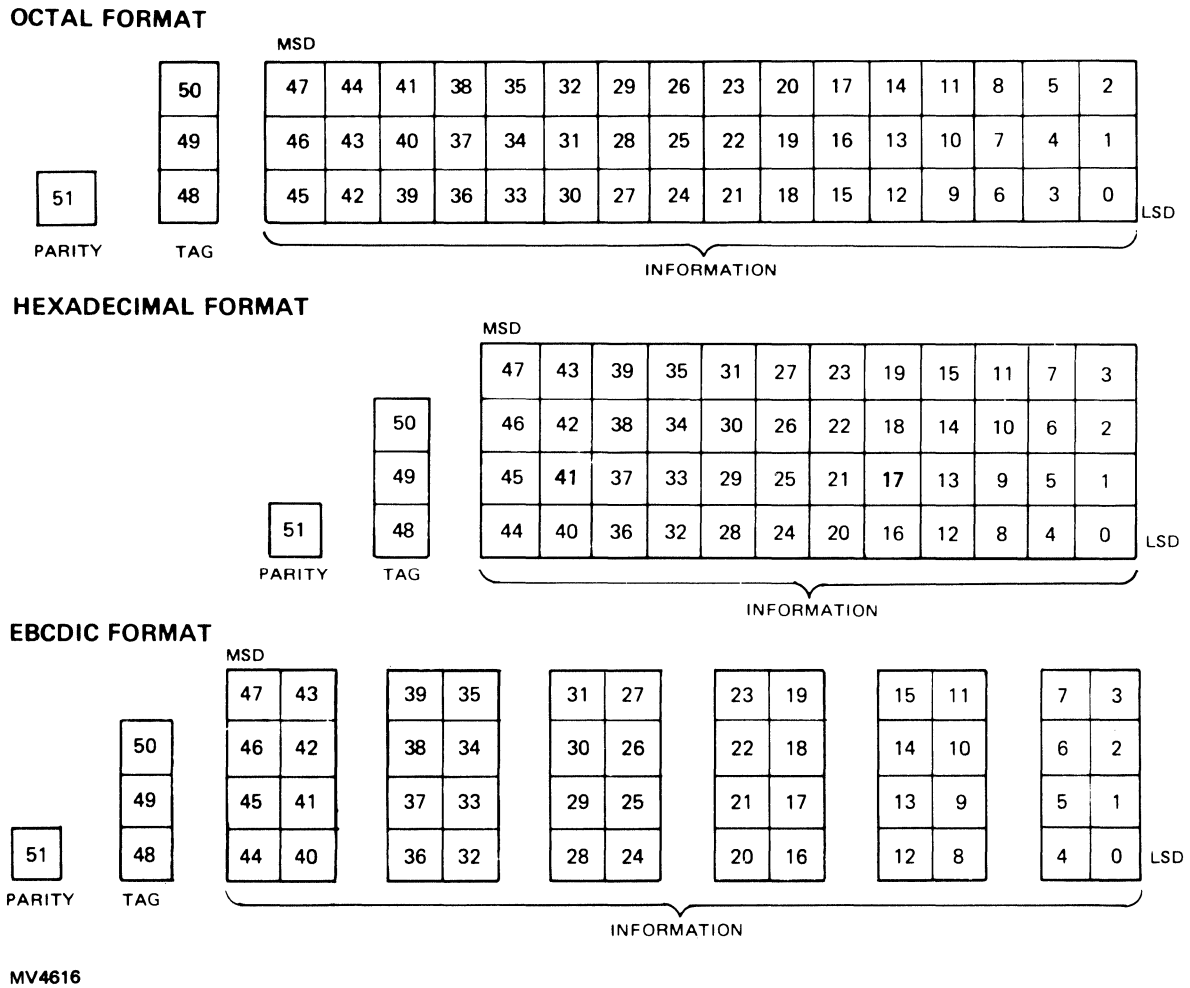


Figure 2-3. B 5900 Word Formats

SUPPORTED DATA TYPES

Supported data types are words that are distinguished by operator code and therefore interpreted according to the data type definition.

Words have 3 tag bits and 48 information bits. The information bits are numbered from 0 to 47, starting with the low-order bit. A field of a word is denoted [first:length]. First is the bit number of the high-order bit of the field (first is less than or equal to 47), and length is the field length in bits (length is less than or equal to 48).

Dynamic fields are wrapped around from the lowest-order bit to the highest-order bit. If the field length is greater than the first bit down to the remaining lower order bits, the field is continued starting from the highest-order bit (47). The following sequence illustrates the field [first:length] in the case where length exhausts the remaining lower-order bits (length > first + 1):

```
47,46, ... ,first, ... ,0,47,46, ... ,last, ... ,1
          <       - length -       >
```

Word types are distinguished by tag value and frequently by additional type bits in the word. The following paragraphs define the data type name, tag, type bit identification, field interpretation, and semantics of each word type.

In this manual, word types are referred to by type name. The data type, double-precision, consists of two words. The term, "item", is used instead of "word" to refer to an entity whose type is variable.

DATA WORDS

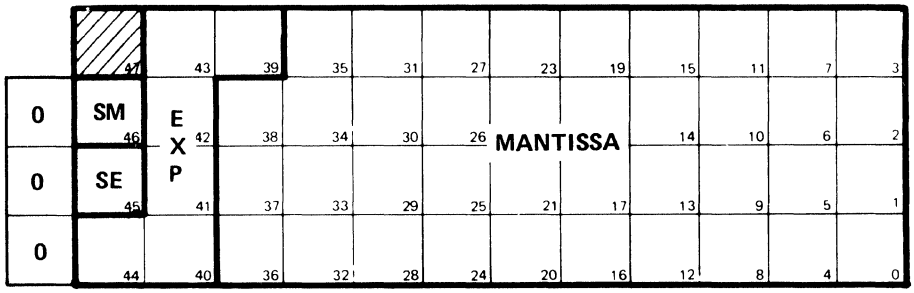
A data word has an even tag value. Data words are computational arguments that do not provide reference or control values to operators.

An important aspect of data words is that they can be stored in memory by Normal Store (as opposed to Overwrite) operations. All odd tagged words are protected from Normal Writes.

Operands (Single-Precision and Double-Precision)

The great majority of data words dealt with by programs are operands. There are two operand types: single-precision which are single words with a tag of 0, and double-precision which are pairs of consecutive words, both with a tag of 2. Single-precision operands and double-precision operands are both referred to as operands.

Neither operand type has a unique interpretation applied to it. Operators, according to their function, apply different interpretations depending on operand characteristics. For example, operands are interpreted as numeric values by arithmetic operators, as bit vectors by word manipulation operators, and as character sequences by string operators. See Operator Definitions for operand interpretations that are applied by each operator group. Figures 2-4 through 2-6 are operand word formats.

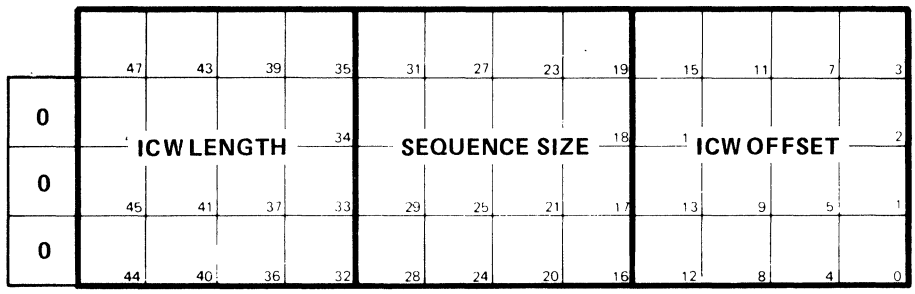


MV4617

Single Precision Operand, Arithmetic Interpretation (Tag of 0)

	[47: 1]	Not used
SM	[46: 1]	Mantissa sign (0=positive, 1=negative)
SE	[45: 1]	Exponent sign (0=positive, 1=negative)
EXP	[44: 6]	The power of 8 to which the mantissa is scaled
MANTISSA	[38:39]	The magnitude of the number before scaling

Figure 2-4. Single Precision Operand Format, Arithmetic Interpretation

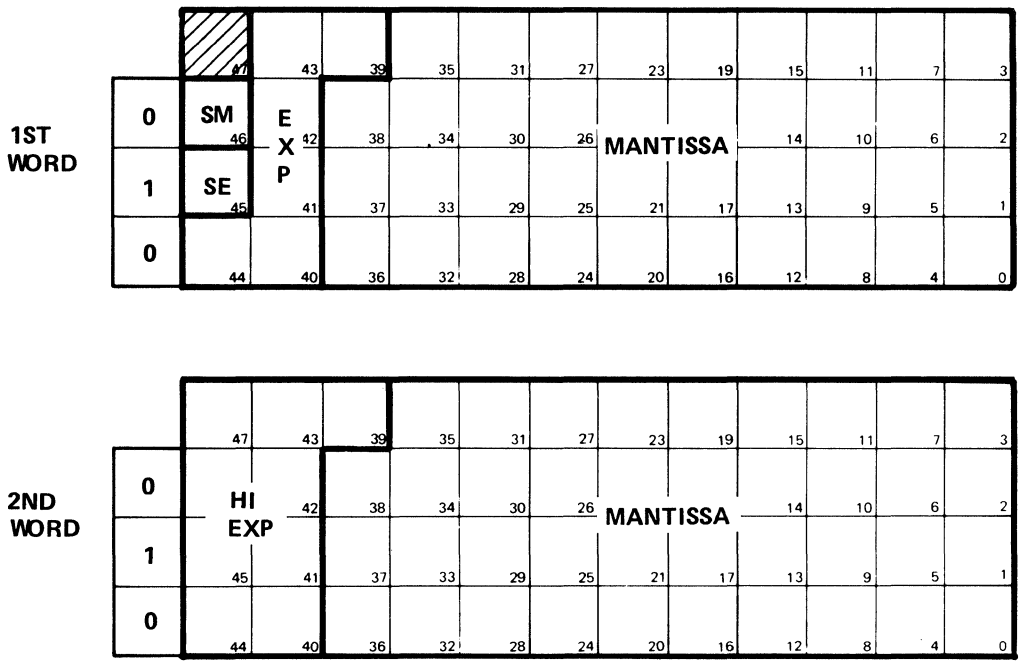


MV4618

Single Precision Operand, Index Control Word Interpretation (Tag of 0)

ICW LENGTH	[47:16]	The length of an element in the sequence
SEQUENCE SIZE	[31:16]	The number of elements in the sequence
ICW OFFSET	[15:16]	The offset from the base of the record to the start of the sequence

Figure 2-5. Single Precision Operand Format, ICW Interpretation



MV4619

Double Precision Operand, Arithmetic Interpretation

		First Word
	[47: 1]	Not used
SM	[46: 1]	Mantissa sign (0=positive, 1=negative)
SE	[45: 1]	Exponent sign (0=positive, 1=negative)
EXP	[44: 6]	The low order 6 bits of the exponent
MANTISSA	[38:39]	The integral portion of the mantissa
		Second Word
HI EXP	[47: 9]	The high order 9 bits of the exponent
MANTISSA	[38:39]	The fractional portion of the mantissa

Figure 2-6. Double Precision Operand Format, Arithmetic Interpretation

Tag 4 Words

Tag 4 words are data words, but the only interpretation applied to them is as a 48-bit vector by a class of computational operators.

Tag 4 words are not fetched normally to the expression stack as operands, and they are not valid arguments for arithmetic computational operators. However, they may be stored in memory by Normal Store operators. B 5900 treatments of tag 4 and tag 6 data words are identical.

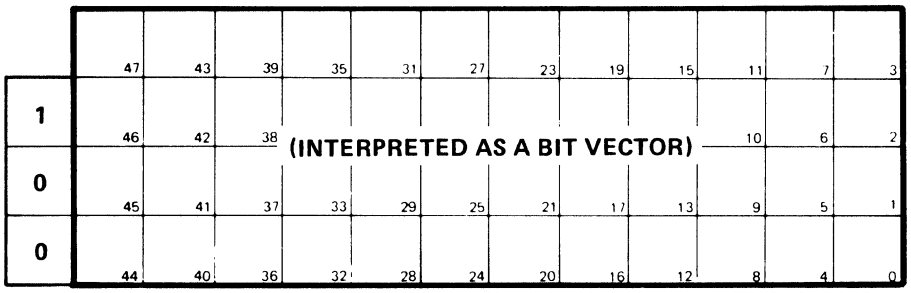


Figure 2-7. Tag 4 Word Format

Uninitialized Operand

Uninitialized operands are words that have tag values of 6. Like Tag 4 Words, they are interpreted as 48-bit vectors by a class of computational operators, but no other interpretation is applied.

Uninitialized operands are not fetched normally to the expression stack as operands, and they are not valid arguments for arithmetic computational operators. However, they can be stored over by normal store operators.

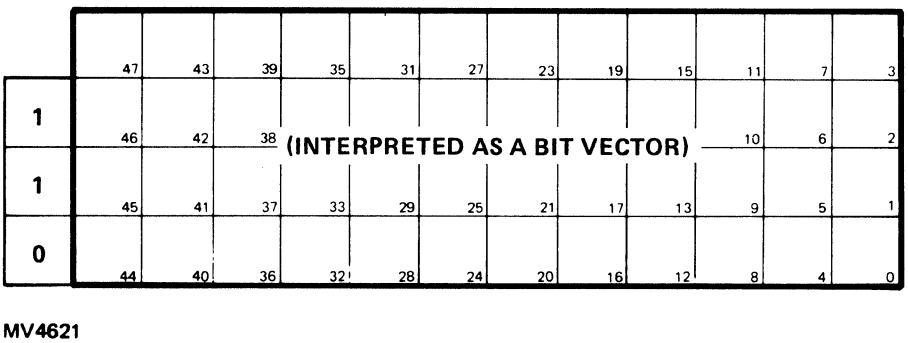


Figure 2-8. Uninitialized Operand Format

PROGRAM CODE WORDS

Variable length operator sequences are stored in arrays of program code words called code segments. Each program code word possesses six 8-bit containers called syllables, numbered 5 to 0 from high-order to low-order. (The mapping of each operator into syllables is specified in Section 4.) Program Code Words have a tag value of 3.

DESCRIPTORS

Memory is organized into variable size segments (arrays) that are either data segments or program code segments. Specially treated arrays such as stacks and Segment Dictionaries are important examples of data segments.

Data Segment Descriptor

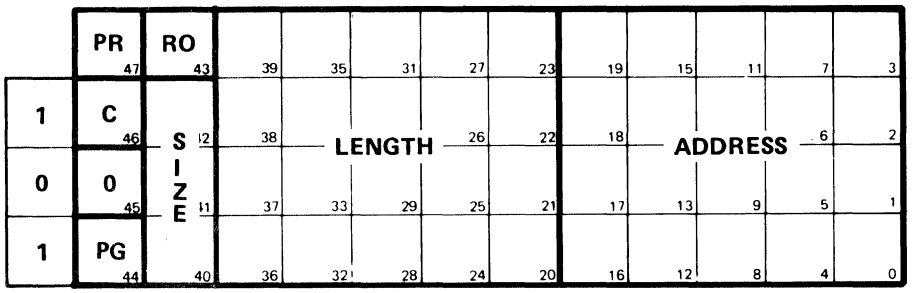
A data segment is an array of elements. An element is either a single word, a double word pair, or a "subword" character requiring 4 or 8 bits. Data Descriptors (DDs) are words that describe data segments. Data Descriptor fields contain the static and dynamic characteristics of the data segment that are required for its management. Data Descriptors have a tag value of 5 and an indexed bit of 0.

The element-size field of the DD specifies the type of array element: single-precision, or double-precision; EBCDIC (8-bit), or hex (4-bit). The length field contains the number of elements in the array. The length of the array in words can be deduced from element-size and length. The term "Word DD" is used for Descriptors whose element-size values are single-precision or double-precision. The term "string DD" is used for Descriptors whose element-size values are EBCDIC or hex.

Memory management of the data segment utilizes a presence bit, an address field, and a copy bit. The copy bit distinguishes two classes of DDs: Mom (original) and copy Descriptors. If the array is present, the address field of both classes is the base memory location of the data segment. The address of a Mom contains a software encoded value referencing the data segment in virtual memory, however the address of a copy Descriptor references a Mom Descriptor for the array.

Generally, there is one Mom DD for an array, and copies are created by most of the operators which fetch DDs to the top of the stack. However, functional operator definition does not require precisely one Mom DD for each array, and a Mom DD can be brought to the top of the stack without being transformed into a copy.

A data segment may be paged, in which case it consists of a number of 256-word pages–the last page of which may be smaller. The Descriptor to a paged data segment (called a paged Descriptor) has a paged bit of 1. If a paged Descriptor is present, its address field references a data segment consisting of Descriptors called page Descriptors to the individual pages of the array. When a paged Descriptor is indexed, another level of indexing must be performed so that the resulting indexed Descriptor references the specified element of the specified page. (See the INDX operator definition for a discussion of the indexing of paged Descriptors).



MV4622

Data Descriptors (Tag value is 5 and indexed bit is 0)

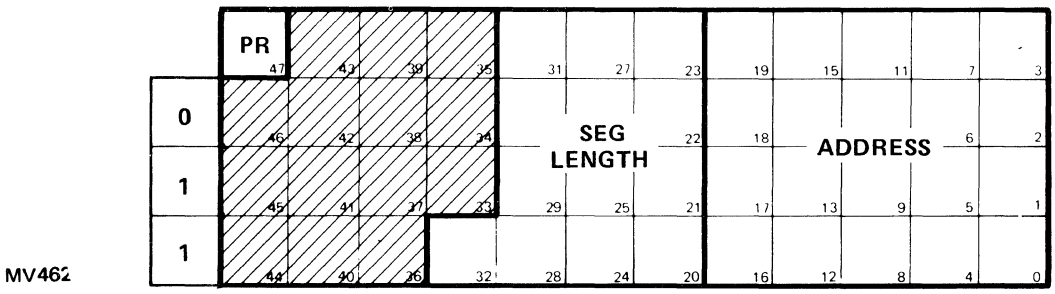
PR	[47:1]	Present bit (0=absent, 1=present)
C	[46:1]	Copy bit (0=Mom, 1=copy)
0 (index)	[45:1]	Indexed bit (0: unindexed)
PG	[44:1]	Paged bit (0=non-paged, 1=paged)
RO	[43:1]	Read only bit (0=Read Write, 1=Read only)
SIZE	[42:3]	The type of array element (0=single-precision, 1=double-precision, 2=hex, 4=EBCDIC, 3,5,6,7 are invalid)
LENGTH	[39:20]	The number of elements in the array
ADDRESS	[19:20]	Address interpretations and their contexts: 1) present: the memory address of the base word of the data segment; 2) absent and copy: the memory address of the associated Mom Descriptor; 3) absent and Mom: a software encoded secondary memory address

Figure 2-9. Data Descriptor Word Format

Code Segment Descriptor

A code segment is a stream of program code words referenced by a code segment Descriptor. The term segment Descriptor is used for code segment Descriptor and data Descriptor for data segment Descriptor. The tag value of a segment Descriptor is 3.

The segment length field contains the number of code words in the segment. Memory management utilizes a presence bit and the address field. If the code segment is present, the address field contains the base memory location of the segment, otherwise, it contains a software encoded secondary memory address.



MV462

Segment Descriptor (tag value is 3)

PR	[47:1]	Present bit (0=absent, 1=present)
	[46:14]	Reserved for future implementation
SEG LENGTH	[32:13]	The number of code words in the segment
ADDRESS	[19:20]	Address interpretations and their contexts are: present, the memory address of the base word of the data segment; absent, a software encoded secondary memory address

Figure 2-10. Segment Descriptor Word Format

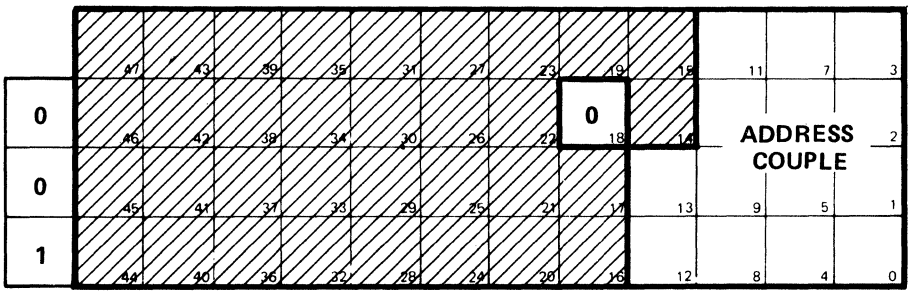
INDIRECT REFERENCES

There are four indirect reference data types: Normal Indirect Reference Words (NIRWs) and Stuffed Indirect Reference Words (SIRWs), which point to locations in the addressing environment; Indexed Data Descriptors (Indexed DDs), which point to individual elements of data segments; and Program Control Words (PCWs), which provide code stream pointers and initial execution state values.

NIRW (Normal Indirect Reference Word)

An NIRW is a dynamic address couple form which references a location in the current addressing environment. An NIRW has a tag value of 1, and bit 18 must be 0.

The only field in an NIRW is an encoded address couple, whose dynamic interpretation yields values for Lambda and Delta. Lambda specifies a lexical level in the current addressing environment. Delta is the offset to the referenced location from the base of the activation record at level Lambda. The location referenced by an NIRW can vary according to the addressing environment at the time of its interpretation.



MV4624

NIRW (tag is 1 and bit 18 is 0)

	[47:29]	Not used
0 (type bit)	[18:1]	Constant value 0
	[17:4]	Not used
ADDRESS COUPLE	[13:14]	The address couple of the referenced location in the current addressing environment.

Figure 2-11. Normal Indirect Reference Word Format

Address couples are encoded in 14 bits with a "floating fence" between Lambda and Delta. Taking advantage of the fact that Lambda must be less than or equal to the lexical level (LL), the number of high-order bits interpreted as the Lambda value varies with the value of ll at evaluation time. The remaining low-order bits are interpreted as the Delta value. The following table gives explicit ranges:

ll Range	Bits Left of Fence	Lambda Range	Delta Range
(0,1)	1	(0 to LL)	(0 to 2^{12})
(2,3)	2	(0 to LL)	(0 to 2^{11})
(4 to 7)	3	(0 to LL)	(0 to 2^{10})
(8 to 15)	4	(0 to LL)	(0 to 2^{9})

The Lambda value is the reverse of the bits to the left of the fence, and the Delta value is taken from the bits to the right of the fence. Following are examples of address couple interpretation. Each pair is the same address couple, but notice the effect of the dynamic fence, indicated by the colon (:).

1. First Pair

   a. at LL= 1, 1:0000000010011 => (1,19)
   b. at LL=13, 1000:0000010011 => (1,19)

2. Second Pair

   a. at LL= 5, 101:00001000000 => (5,64)
   b. at LL= 3, 10:100001000000 => (1,2112)

SIRW (Stuffed Indirect Reference Word)

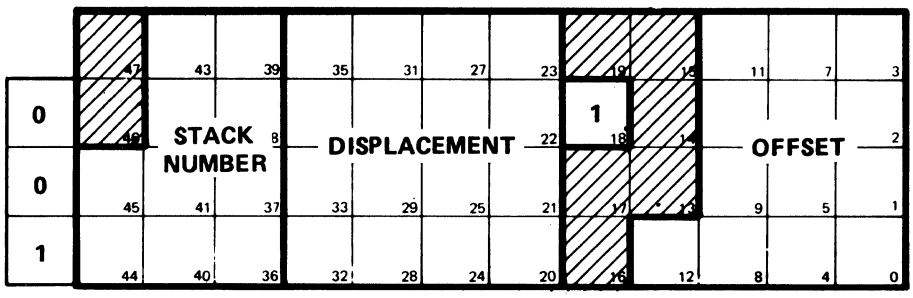
An SIRW, like an NIRW, references a location in an addressing environment. The form of the reference is such that an SIRW always points to the same location, regardless of the state of the current addressing environment. SIRWs have a tag value of 1, and bit 18 must be 1. (Bit 18 is 0 for an NIRW).

An SIRW has three fields: stack number, displacement, and offset. The memory location referenced by an SIRW is computed by the following function:

Base address (stack number) + displacement + offset.

Base address (stack number) is the address of the bottom of the stack. Base + displacement yields the address of the base word of an activation record, and offset is the index of the referenced location relative to that base.

Note that the Base + displacement expression is a Lexical Link, and offset corresponds to NIRW Delta.



MV4625

SIRW (Tag value of 1 and bit 18 is 1)

	[47:2]	not used
STACK NUMBER	[45:10]	The identification of the stack containing the referenced location
DISPLACEMENT	[35:16]	The displacement from the base of the stack to the base of the activation record
	[19:1]	not used
(1) type bit	[18:1]	Constant value 1
	[17:5]	not used
OFFSET	[12:13]	The offset from the base of the activation record to the referenced location

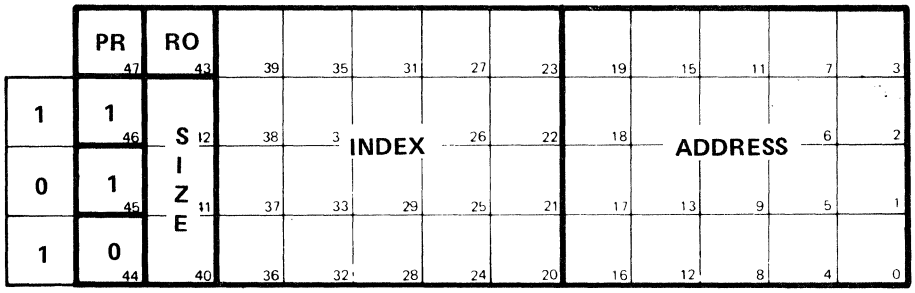
Figure 2-12. Stuffed Indirect Reference Word Format

Indexed DD (Indexed Data Descriptor)

Indexed DDs reference an individual element of a data segment. The interpretation of an Indexed DD is a variation of the interpretation of a DD. The tag of an Indexed DD is 5, and its Indexed bit is 1.

An Indexed DD is a copy and cannot be paged. Its element-size, presence bit, and address field are interpreted identically to those of a DD. The term Indexed Word DD is used for Indexed DDs whose element-size values are single-precision or double-precision, and Indexed String DD (or Pointer) is used for Indexed DDs whose element-size values are EBCDIC or hex.

The interpretation of the index to the referenced element depends on element-size. For Indexed Word DDs, the index field is the word index from the base of the array to the single-precision word or to the first word of the double-precision word pair. For Pointers, the index field consists of two sub-fields: word index, the index from the base of the array to the word containing the referenced character, and char index, the character index within the word. For EBCDIC Pointers, char index must be in the range (0 to 5), and for hex Pointers, it must be in the range (0 to 11). Char index 0 is the highest-order character in the word.



Indexed DD (tag is 5 and indexed bit is 1)

PR	[47:1]	Presence bit (0=absent, 1=present)
copy	[46:1]	Copy bit (always 1; denotes a copy)
indexed	[45:1]	Indexed bit (always 1; denotes indexed)
paged	[44:1]	Paged bit (always 0; denotes non-paged)
RO	[43:1]	Read only bit (0=Read Write, 1=Read Only)
SIZE	[42:3]	The type of array element (0=single-precision, 1=double-precision, 2=hex, 4=EBCDIC, and (3,5,6,7) are invalid values)
INDEX	[39:20]	Used if element-size is single or double-precision: the word index from the base of the array to the referenced word
char index	[39:4]	Used if element-size is EBCDIC or hex: the index within the word of the referenced character
word index	[35:16]	Used if element-size is EBCDIC or hex: the index from the base of the array to the word containing the referenced character
ADDRESS	[19:20]	Address interpretations (and their contexts) are: 1) present, the memory address of the base word of the array; 2) absent, the memory address of the associated mom data Descriptor

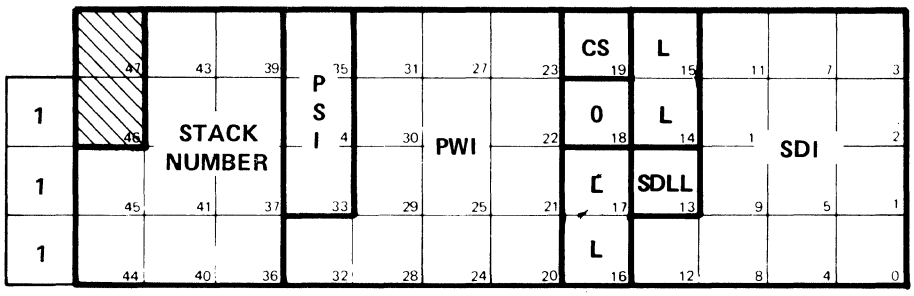
Figure 2-13. Indexed Data Descriptor Word Format

PCW (Program Control Word)

A Program Control Word (PCW) contains the initial code stream pointer and execution state values associated with an activation record in the program. A PCW establishes the execution state for an activation record when it is entered (when it becomes the topmost activation record). The PCW has a tag value of 7.

The PCW code stream pointer consists of the fields SDLL, SDI, PWI and PSI. (See Figure 2-14.) The LL field of the PCW indicates the lexical level at which the activation record is to run. The control state attribute specifies execution in normal or control state.

A PCW also contains a stack number field, the identification of the stack in which the PCW is stored. Stack number is not required by the operator set.



MV4627

PCW (tag value is 7)

	[47:2]	not used
STACK NUMBER	[45:10]	The identification of the stack in which the PCW is stored
PSI	[35:3]	The Program Syllable Index code stream pointer component
PWI	[32:13]	The Program Word Index code stream pointer component
CS	[19:1]	The initial value of the Control State Boolean (0=normal state, 1=control state)
INVALID LL	[18:1]	Must be zero
LL	[17:4]	The lexical level at which the activation record runs
SDLL	[13:1]	The Segment Dictionary lexical level code stream pointer component
SDI	[12:13]	The Segment Dictionary Index code stream pointer component

Figure 2-14. Program Control Word Format

STACK LINKAGE WORDS

There are three data types utilized for stack linkage. An MSCW (Mark Stack Control Word) and an RCW (Return Control Word) are the two words which contain stack linkage values for an activation record in the addressing environment. A TSCW (Top of Stack Control Word) is used to preserve processor state in a stack in which there is no current processor activity.

There are five word types that have a tag of 3; These are the three stack linkage words, Segment Descriptors, and Program Code Words. There are not any type bits within the words, which based only on tag value are not distinguishable from each other. The five word types are distinguished by context. The integrity of execution and addressing environment state depend on this.

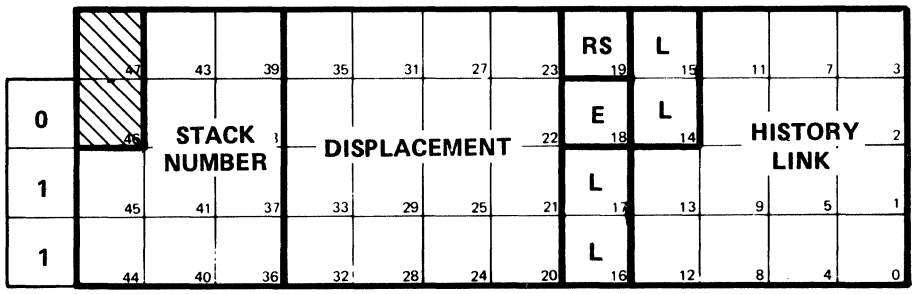
MSCW (Mark Stack Control Word)

A Mark Stack Control Word (MSCW) contains the History Link and Lexical Links for an activation record. The MSCW is the base word in the activation record. All links to the MSCW point to it.

The ll field indicates the lexical level of the activation record containing the MSCW. The History Link is a displacement down the stack to the prior activation record. The entered bit indicates whether the activation record is inactive (containing only a History Link) or entered (fully linked into the addressing environment).

Two fields comprise the Lexical Link pointer to the activation record which is the immediate global addressing space. Stack number is the identification of the stack containing the activation record. Displacement is the offset from the base of the stack number stack to the base of the activation record. (Note that an SIRW contains stack number and displacement fields with the same semantics).

The restart bit is the value to which the processor RS Boolean will be set when the current activation record is exited.



MV4628

MSCW (tag value of 3)

	[47:2]	not used
STACK NUMBER	[45:10]	Identifies the stack containing the activation record to which the Lexical Link points
DISPLACEMENT	[35:16]	The displacement down the stack number stack from the base of the activation record addressed by the Lexical Link to the base of the stack
RS	[19:1]	The value to which the RS (Restart) Boolean will be set when the activation record is exited
E	[18:1]	The entered bit (0=inactive, 1=entered)
LL	[17:4]	The lexical level at which the activation record runs
HISTORY LINK	[13:14]	The displacement down the containing stack from the MSCW to the base of the prior (initiating) activation record

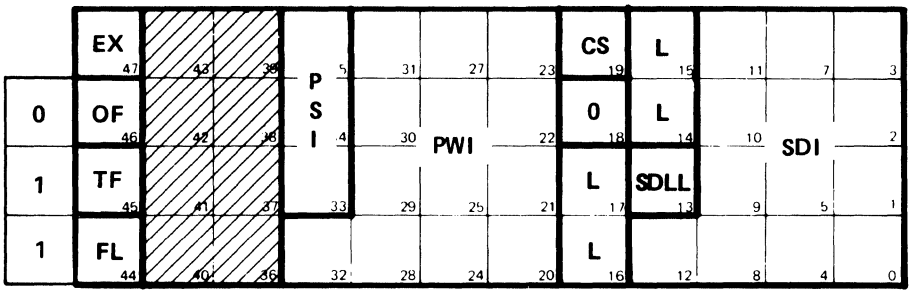
Figure 2-15. Mark Stack Control Word Format

RCW (Return Control Word)

An RCW is stored directly above the MSCW at the base location plus one of an entered activation record. The RCW preserves code stream pointer and execution state to be restored when the activation record is exited. An RCW has a tag value of 3.

The RCW code stream pointer consists of the fields SDLL, SDI, PWI and PSI. Preserved execution state consists of control state, the CS Boolean, the 4 processor state Booleans (defined in General Boolean Accumulators), and LL, the lexical level of the activation record initiating this activation record.

The True False Occupied Flip-Flop (TFOF) is not required in the B 5900. General operator restart capability is provided by the RS (restart) Boolean, which is preserved in the MSCW.



MV4629

RCW (tag value of 3)

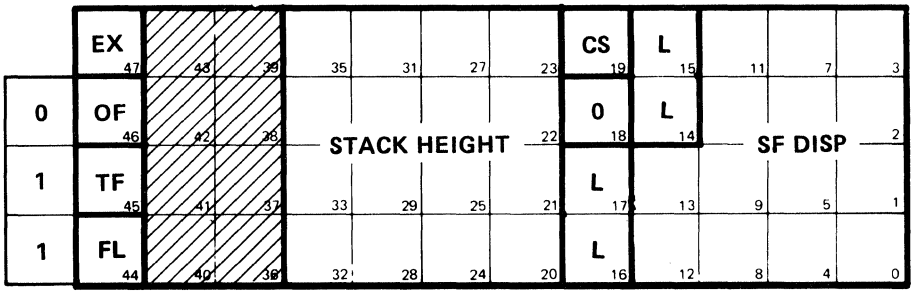
EX	[47:1]	External Sign Flip-Flop
OF	[46:1]	Overflow Flip-Flop
TF	[45:1]	True False Flip-Flop
FL	[44:1]	Float Flip-Flop
	[43:8]	not used
PSI	[35:3]	The Program Syllable Index code stream pointer component
PWI	[32:13]	The Program Word Index code stream pointer component
CS	[19:1]	The Control State Boolean
invalid II	[18:1]	Constant value 0
LL	[17:4]	The lexical level of the activation record which initiated this activation record.
SDLL	[13:1]	The Segment Dictionary lexical level code stream pointer component
SDI	[12:13]	The Segment Dictionary Index code stream pointer component

Figure 2-16. Return Control Word Format

TSCW (Top of Stack Control Word)

A TSCW preserves processor state in a stack where no current processor activity exists (inactive stack). The tag of a TSCW is 3.

The S register (the pointer to the top expression in the current stack) and the F register (the current MSCW pointer) are preserved in specific fields of the TSCW. Stack height holds the displacement from the base of the stack up to S. SF disp holds the displacement down the stack from S to F. LL indicates the lexical level of the topmost activation record when the stack was last active. The processor state Booleans and the CS Boolean are also preserved in the TSCW.



MV4630

TSCW (tag value of 3)

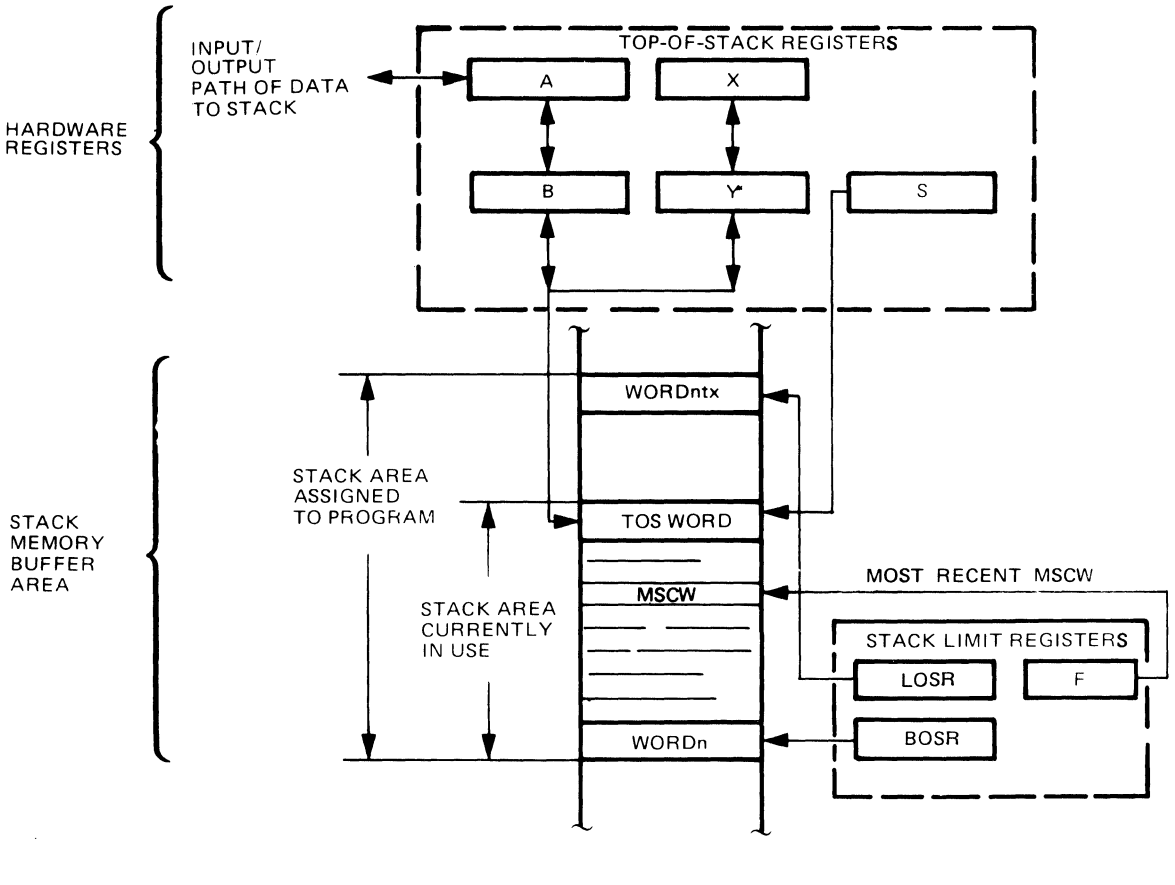
EX	[47:1]	External Sign Flip-Flop
OF	[46:1]	Overflow Flip-Flop
TF	[45:1]	True False Flip-Flop
FL	[44:1]	Float Flip-Flop
	[43:8]	not used
STACK HEIGHT	[35:16]	The displacement from the base of the stack to the top of the expression stack
CS	[19:1]	The CS Boolean
invalid II	[18:1]	Constant value 0
LL	[17:4]	The lexical level of the topmost activation record when the stack was last active
SF DISP	[13:14]	The displacement from the top of the expression stack down to the historical chain pointer (F)

Figure 2-17. Top of Stack Control Word Format

SECTION 3
STACK CONCEPT AND REVERSE POLISH NOTATION

THE STACK

The stack is the memory storage area assigned to a job, and provides storage for the basic program and data references for the job. The stack also provides for temporary storage for data and job history. When a job is activated, registers A, B, X, and Y, which are located in the Data Processor module, are linked to memory locations of the stack of the job (see Figure 3-1). This linkage is established by the Stack Pointer Register (S register), which contains the memory address of the last word placed in the stack. The four top-of-stack registers (A,B,X, and Y) extend the stack to provide quick access for data manipulation. Another stack pointer value (the F register) always points to the most recent MSCW in the stack.



MV4631

Figure 3-1. Top of Stack and Stack Bounds Registers

The number of words in the memory portion of the stack is equal to the difference between the values of the S register, and BOSR (S minus BOSR). Data are brought into the stack through the top-of-stack registers in such a manner that the last word placed in the stack (as indicated by the value of the S register) is the first word to be extracted from the stack (last in first out method). The total capacity of the top-of-stack registers is two words or two operands. Loading a third word into the top-of-stack registers causes the first word to be pushed from the top-of-stack into the memory portion of the stack. The stack pointer value in the S register is incremented by one as a word or operand is pushed into memory, and is decremented by one when a word operand is withdrawn from the stack. As a result, the S register continually points to the last word placed into the memory portion of the job stack.

BASE AND LIMIT OF STACK

The stack of a job is bounded, for memory protection, by two registers: the base-of-stack register (BOSR) and the limit-of-stack register (LOSR). The contents of BOSR define the base of the memory portion of the stack, and the contents of LOSR define the upper limit of the memory portion of the stack. The job is interrupted if the S register is set to a value that is present in either LOSR or BOSR. If the S register equals or exceeds the value of LOSR, a stack overflow interrupt occurs.

BIDIRECTIONAL DATA FLOW IN THE STACK

The contents of the top-of-stack registers are maintained by the microprogram to meet the requirements of the current operator. If the current operator requires data transfer into the memory portion of the stack, the top-of-stack registers receive the incoming data, and surplus contents in the top-of-stack registers are pushed down into the memory portion of the stack. "Pushing" data into the memory means that the bottom word or operand in the top-of-stack register is transferred to the next word or operand in sequence in the memory portion of the stack. Pushing data down into the memory portion of the stack makes room in the top-of-stack registers to contain the incoming data that is required by the current operator.

Data is also brought automatically from the memory portion and placed in the top-of-stack registers when the operator requires that the top-of-stack registers be filled. This function, called a pop function, is the opposite of the push function described in the previous paragraph. A pop transfers the last operand or word in the memory portion of the stack into the second word position in the top-of-stack registers. The word or operand in stack memory is then deleted by decrementing the S register. The "push", and "pop" functions, described in the following paragraphs, comprise the maintenance of the top-of-stack registers.

Stack Push

A stack push occurs when a third word or operand is loaded into the top-of-stack registers, and both the A register and B register already contain stack words or operands. A push consists of moving data from the top-of-stack registers to the local memory portion of the stack. Moving data to the local memory portion of the stack makes room in the top-of-stack registers so that a third operand may be loaded into the top-of-stack registers.

Stack Pop

A stack pop up occurs when an operand or word is moved from local stack memory to the top-of-stack portion of the stack. A pop can only occur when a machine operator is executed by the DP. The operator which is to be performed requires that words or operands be present in the top-of-stack registers, and that such words or operands may not be present in the proper top-of-stack registers.

DOUBLE-PRECISION STACK OPERATION

The top-of-stack registers are operand-oriented rather than word-oriented. Therefore, calling a double-precision operand into the top-of-stack registers causes two memory words to be loaded into the top-of-stack registers. The first word is loaded into the B register, where the tag bits are checked. If the value indicates double-precision, the second word is loaded into the Y register. The two registers are concatenated, or linked together, to form the double-precision operand. A double-precision operand located in the B and Y registers reverts to two words when pushed down into the stack memory.

TOP-OF-STACK REGISTER CONDITIONS

Two logical indicators are used to exhibit the condition of the top-of-stack registers. These two indicators are the A Register Occupied Flip-flop (AROF), and the B Register Occupied Flip-flop (BROF). The interpretations of these two flip-flop indicators are as follows:

AROF	BROF	Meaning
0	0	Neither the A, nor the B register contains valid data. The top word in the stack is presently located in the local memory address specified by the contents of the S register.
0	1	The B register contains the top word in the stack, and the contents of the A register are not valid. The second word in the stack is presently located in the local memory address specified by the contents of the S register.
1	0	The A register contains the top word in the stack, and the contents of the B register are not valid data. The second word in the stack is presently located in the local memory address specified by the contents of the S register.
1	1	The A register contains the top word in the stack, and the second word in the stack is presently in the B register. The third word in the stack is in the local memory address specified by the contents of the S register.

STACK ADJUSTMENTS

Each machine operator that is executed by the processor contains the requirements to adjust the top-of-stack registers so that they contain the appropriate data for the required operation. The following conventions are used to show what stack adjustment is required:

(ADJ 0,0)

Both the A and B registers are to be adjusted so that their contents are not valid. The top word in the stack is to be located in the local memory address specified by the contents of the S register.

The processor will use the state of the AROF and BROF flip-flops to determine if the stack must be pushed down to achieve the required adjustment. The 0,0 portion of the convention notation shows what the logical states of AROF and BROF must be to satisfy the requirements of the adjustments. The first 0 in the expression of the notation defines what the logical state of the AROF flip-flop must be at the conclusion of the stack adjustment. The second 0 in the expression defines what the logical state of the BROF flip-flop must be at the conclusion of the stack adjustment. The ADJ portion of the convention notation reads "adjust the stack until AROF and BROF meet the logical states ...".

(ADJ 0,1)

The A register is to be adjusted so that its contents are not valid. The top word or operand in the stack is to be present in the B register, and the second word or operand in the stack is to be located in the local memory address specified by the contents of the of the S register.

(ADJ 1,0)
The A register is to be adjusted so that it contains the top word or operand in the stack. The B register must not contain valid data. The second word or operand is located in the local memory address specified by the contents of the S register.

(ADJ1,1)
The A register is to be adjusted so that it contains the top word or operand in the stack. The B register is to be adjusted so that it contains the second word or operand in the stack. The third word or operand in the stack is to be in the local memory address pointed at by the contents of the S register.

(ADJ 0,2)
The A register is to be adjusted so that its contents are not valid. The B register condition is immaterial to the operation. The top word in the stack is present in the B register if BROF is set.

(ADJ 1,2)
The A register is to be adjusted so that it contains the top word in the stack. The B register condition is immaterial to the operation. The second word in the stack is located in the B register if BROF is set.

(ADJ 1,3)
The A register is adjusted so that it contains the top word in the stack only when the original stack condition is AROF reset and BROF reset (0,0). If there is any original condition other than (0,0), no stack adjustment will occur.

Some machine operations require that several stack adjustments must be performed during the course of the operation. Such operations merely pause at the appropriate place until the adjustment is completed, and then continue the sequence.

Stack push and stack pop, which were previously defined, are intrinsic functions of the stack adjustments. That is, a push or pop may be implied by the current state of the top-of-stack registers, and by the required stack adjustment. When a stack push or pop is implied, such operations will be performed as an integral and automatic function of the stack adjustment procedure.

DATA ADDRESSING

The B 5900 data processor provides three methods for addressing data or program code:

Data Descriptor (DD)/Segment Descriptor (SD)
Normal Indirect Reference Word (NIRW)
Stuffed Indirect Reference Word (SIRW)

Data Descriptors (DD) and Segment Descriptors are used to address information located outside of stacks. Data Descriptors address data; Segment Descriptors address executable code. NIRWs and SIRWs are used to address information located within stacks. NIRWs address information located within the stack of a job. SIRWs are able to address information located in any stack.

Data Descriptor

In general, the Descriptor describes and locates data associated with a given job. The Data Descriptor (DD) is used to fetch data to the stack or to store data from the stack into an array located outside the stack area of the job. The formats of the Data and Segment Descriptors are given in Section 2. The address field in each of these Descriptors is 20 bits in length; this field contains the absolute address of an array in either local memory or in the disk file, as indicated by the presence bit. If the presence bit is set, the referenced data is in memory; if reset, it is in the disk file.

PRESENCE BIT

A presence bit interrupt occurs when the job references data by means of a Descriptor in which the presence bit is equal to 0. (An indication that the data is located in a disk file rather than in local memory.) The Master Control Program (MCP) recognizes the presence bit interrupt and transfers data from disk file storage to local memory. After the data transfer to local memory is completed, the MCP sets the presence bit in the Descriptor to 1, and places the new local memory address into the address field. The interrupted job is then resumed.

INDEXED BIT

A Data Descriptor describes either an entire array of data words, or a particular element within an array of data words. If the Descriptor describes the entire array, the indexed bit in the Descriptor is 0, indicating that the Descriptor has not yet been indexed. The length field of the Descriptor defines the length of the data array.

Invalid Index

A particular element of an array is described by indexing an array Descriptor. Memory protection is ensured during indexing operations by performing a comparison between the length field of the Descriptor and the index value. An invalid index interrupt results if the index value exceeds the length of the local memory area defined by the Descriptor, or if the index is less than 0.

Valid Index

If the index value is valid, the length field of the Descriptor is replaced by the index value, and the indexed bit in the Descriptor is set to 1 to indicate that indexing has taken place. The address and index fields are added together to generate the absolute machine address whenever an indexed Data Descriptor in which the presence bit is set is used to fetch or store data.

If the element-size of the Descriptor has a value of double-precision the index value is doubled before indexing.

Read-Only Bit

The read-only bit specifies that the local memory area described by the Data Descriptor is a read-only area. If the read-only bit of a Descriptor is set, and the area referenced by that Descriptor is used for storage purposes, an interrupt results.

Copy Bit

The copy bit identifies a Descriptor as a copy of a mom Descriptor and is related to the presence bit action. The copy bit links multiple copies of an absent Descriptor (a condition resulting from the presence bit being off) to the mom Descriptor. The copy bit mechanism is invoked when a copy is made in the stack. If it is a copy of the original absent Descriptor, the processor sets the copy bit and inserts the address of the mom Descriptor into the address field. Thus, multiple copies of absent Data Descriptors are all linked back to the mom Descriptor.

REVERSE POLISH NOTATION

Reverse Polish notation is an arithmetical or logical notation system using only operands and operators arranged in sequence or strings, thus eliminating the necessity of defining the boundaries of any terms. Figure 3-2 presents a flow chart for conversion to reverse Polish notation. Table 3-1 is a guide for generating Polish string from an expression source.

Table 3-1. Simplified Rules for Generating a Polish String

Name	Action
Variable or constant	Place variable or constant in string being built and examine next symbol.
Operator separator "(" or "["	Place in delimiter list and examine next symbol.
Arithmetic or Boolean operator and last-entered delimiter list symbol were as follows: An operator of lower priority. A left bracket "[" or parenthesis "(". A separator. Nothing (delimiter list empty).	Place operator in the delimiter list and examine next source symbol.
Arithmetic or Boolean operator and last-entered delimiter list symbol were as follows: an operator of priority equal to or greater than the symbol in the source.	Remove the operator from the delimiter list and place it in the string being built. Then compare the next symbol in the delimiter list against the source expression symbol.

POLISH STRING

The essential difference between reverse Polish notation and conventional notation is that operators are written to the right of the operands instead of between them. For example, the conventional B + C is written BC+ in reverse Polish notation: A=7x(B+C) becomes A 7 B C + x :=.

Any expression written in reverse Polish notation is called a polish string. In order to fully understand this concept, the user should know the rules for evaluating a polish string.

RULES FOR EVALUATING A POLISH STRING

The following is the procedure for evaluating a polish string:

1. Scan the string from left to right.
2. Note the operands and the order in which they occur.
3. When an operator is encountered perform the following:

    a. Record the last two operands encountered.
    b. Execute the required operation.
    c. Discard the two operands.
    d. Consider the result of (2) as a single operand.

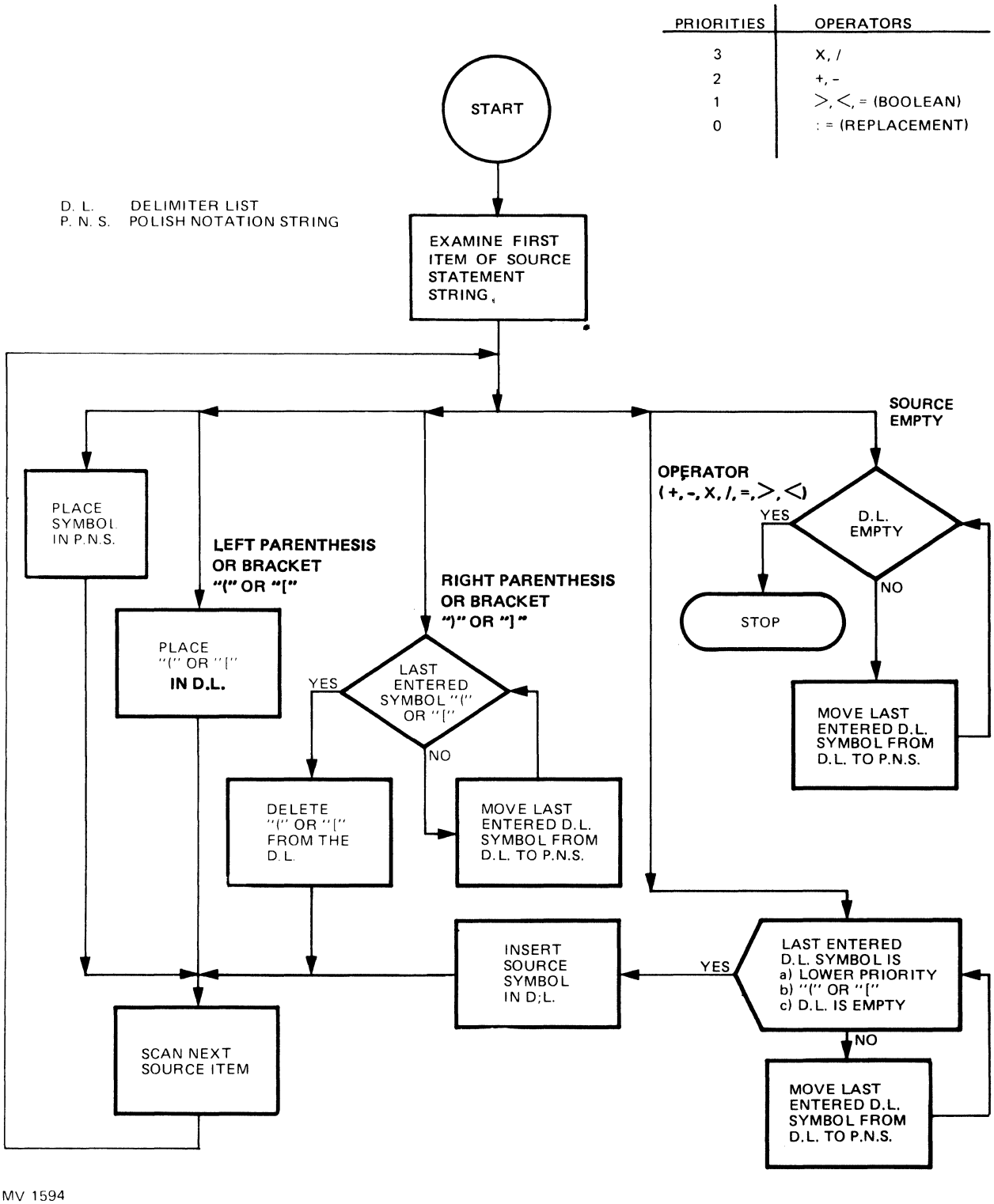


Figure 3-2. Reverse Polish Notation Flow Chart

Following this procedure, the reverse Polish string A 7 B C + x := results in A assuming the value 7 x (B+C) (See Table 3-2).

NOTE

Because replacement operators vary depending upon the language used, ←, =, and := are equivalent for this discussion.

SIMPLE STACK OPERATION

All program information must be in the system before it can be used. Input areas are allocated for information entering the system, and output areas are set aside for information exiting the system; array and table areas are also allocated to store certain types of data. Thus data is stored in several different areas: the input/output areas, data tables (arrays), and the stack. Since all work is done in the arithmetic registers, all information or data is transferred to the arithmetic registers and the stack.

Table 3-2. Evaluation of Polish String A 7 B C + X :=

Step No.	Symbol Being Examined	Symbol Type	Operands Being Remembered Order of Occurrence (1 or 2) Before Operation	Occurring Operation	Operation Results
1	B	Operand			
2	C	Operand	1 B		
3	+	Add Operator	2 C 1 B	B+C	(B+C)
4	7	Operand	1 (B+C)		
5	x	Multiply Operator	2 7 1 x(B+C)	7x(B+C)	7x(B+C)
6	A	Name	1 7x(B+C)		
7	:=	Replace	2 A	A:=7x(B+C)	A=7x(B+C)

At this point, an ALGOL assignment statement and the reverse Polish notation equivalent will be related to the stack concept of operation. The example is Z := Y+2x(W+V) where := means "is replaced by". In terms of a computer program, this assignment statement indicates that the value resulting from the evaluation of the arithmetic expression is to be stored in the location represented by variable Z.

When Z := Y+2x(W+V) is translated to reverse Polish notation, the result is ZY2WV+x+:=. Each element of the example expression causes a certain type of operator to be included in the machine language program when the source problem is compiled. The following is a detailed description of each element of the example, the type of operator compiled, and the resulting operation (see Figure 3-3 and Table 3-3).

In the expression above, Z is to be the recipient of a value, the address of Z must must be placed into the stack just prior to the store command. This is accomplished by a name call operator (NAMC) which places a Normal Indirect Reference Word (NIRW) in the stack. The NIRW contains the address of Z in the form of an "address couple" that references the memory location reserved in the stack for the variable Z.

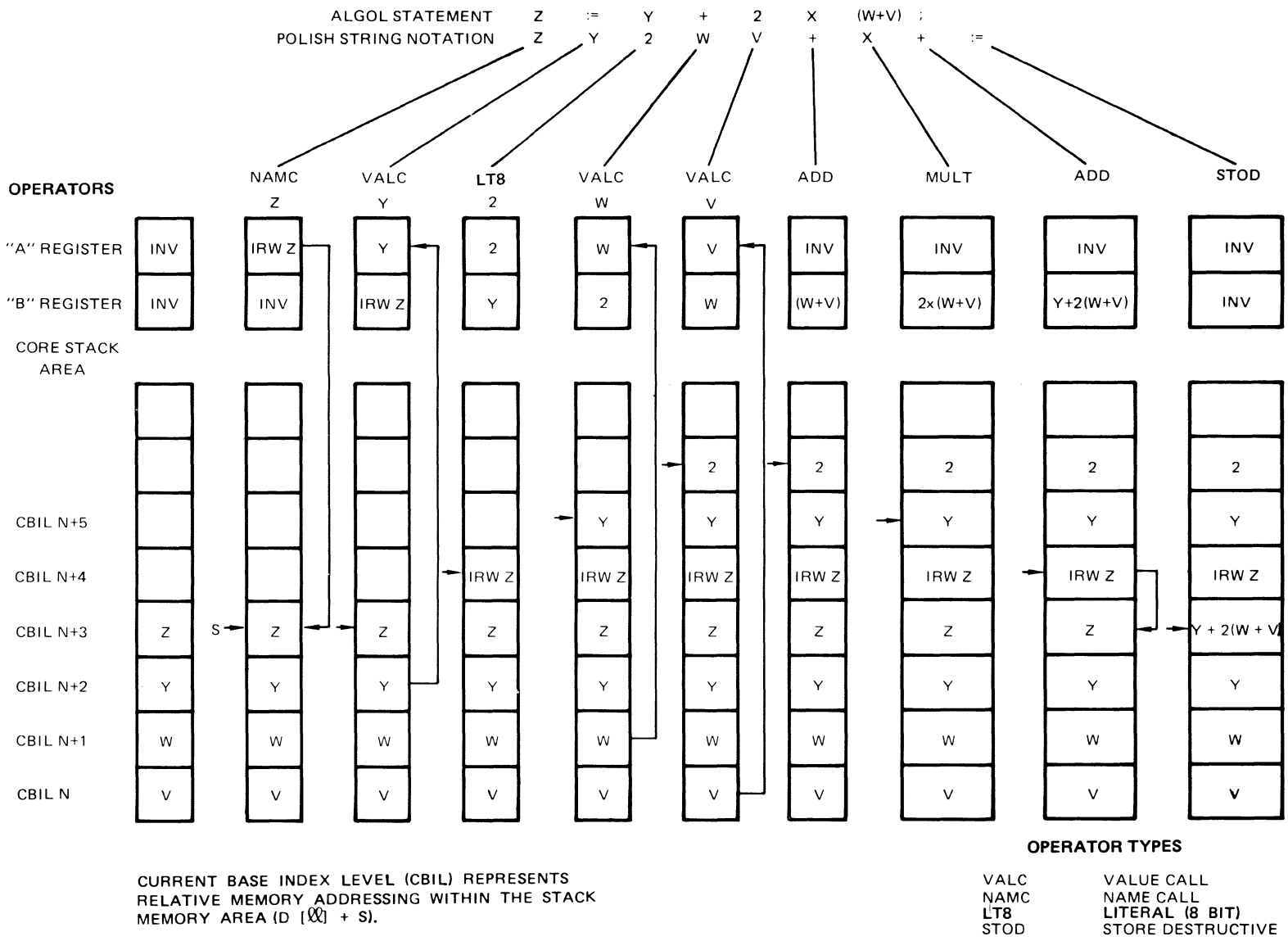


Figure 3-3. Stack Operation

Table 3-3. Description of Stack Operation

Execution Sequence	Reverse Polish Notation Element	Syllable Type Compiled	Function of Syllable During Running of the Program
0			Stack location of program variable illustrated.
1	Z	Name call for Z	Build an IRW that contains the address of Z and place it on the top of the stack.
2	Y	Value call for Y	Place the value of Y in the top of the stack.
3	2	Literal 2	Place a 2 in the top of the stack.
4	W	Value call for W	Place the value of W in the top of the stack.
5	V	Value call for V	Place the value of V in the top of the stack.
6	+	ADD Operator	Add the top two words in the stack and place the sum in the B register as the top of the stack.
7	x	MULT Operator	Multiply the two top-of-stack operands. The product is left in the B register as the top of the stack.
8	+	ADD Operator	Add the top two words in the stack and leave the sum in the B register as the top of the stack.
9	:=	Store Delete (STOD) Operator	Store an item into memory. The address in which to store is indicated by an IRW or a DD; the address can be above or below the stored. Item is removed from the stack.

Since Y is to be added to a quantity, Y is brought into the top of the stack as an operand by way of a value call (VALC) operator that references Y. The value 2 is then brought to the stack, with an eight-bit literal operator (LT8). Since W and V are to be added, the respective variables are brought to the stack with value call operators. The ADD operator adds the two top operands and places the sum in the top of the stack. This example assumes, for simplicity, single-precision operands not requiring use of the X and Y registers which are used in double-precision operations.

The multiply operator is the next symbol encountered in the reverse Polish string; when executed, it places the product "2x(W+V)" in the top of the stack. The next operator, ADD, when executed, leaves the final result "Y+2x(W+V)" in the top of the stack.

The store operator completes the execution of the statement Z := Y+2x(W+V). The store operation examines the two top-of-stack operands for an IRW or Data Descriptor. In this example, the IRW addresses the location where the computed value of Z is to be stored. The stack is empty at the completion of this statement.

PROGRAM STRUCTURE IN LOCAL MEMORY

When a problem is expressed in a source language, portions of the source language fall into one of two categories. One describes the constants and variables that will be used in the program, and the other describes the computations that will be executed (see Figure 3-4). When the source program is compiled, variables are assigned locations within the stack, whereas the constants are embedded within the code stream that forms the computational part. A program residing in memory occupies separately allocated areas. "Separately allocated" means that each part of the program may reside anywhere in memory, and the actual address is determined by the MCP. In particular, the various areas are not assigned to contiguous memory areas. Registers within the processor indicate the bases of the various areas during the execution of a program.

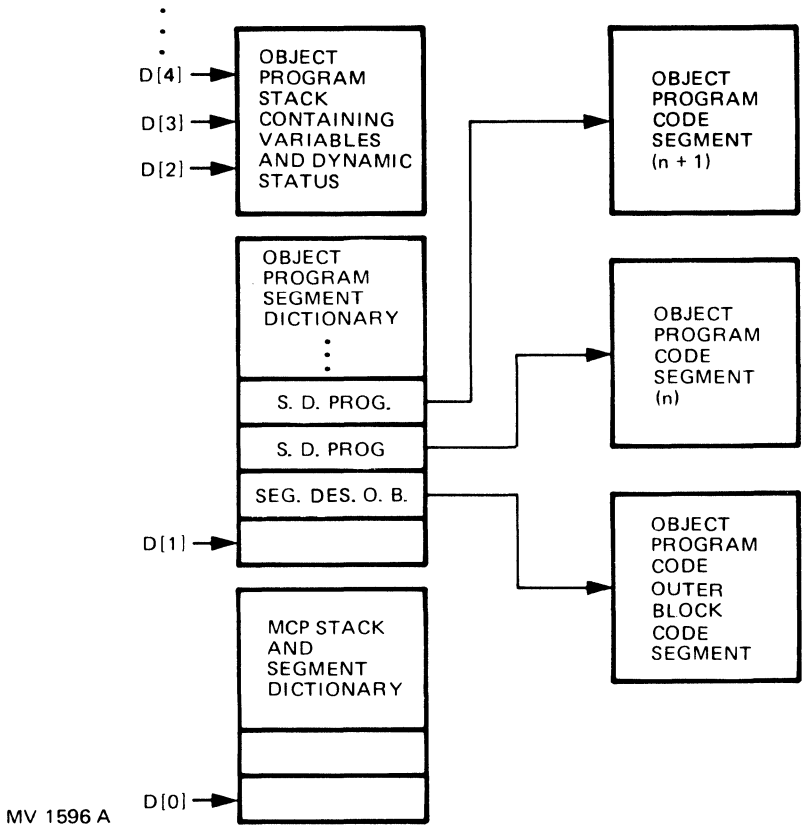


Figure 3-4. Object Program in Memory

LOCAL MEMORY AREA ALLOCATION

The separately allocated areas of a program are as follows:

Program Segments
: These are sequences of instructions (operators) that are performed by the processor in executing the program. Program segments are distinct form data areas in that they contain no data and are not modified by the processor in the execution of the program.

Segment Dictionary
: This is a table containing one word for each program segment. This word tells whether the program segment is in local memory or on disk, and gives the corresponding local memory or disk address of the program segment.

Stack Area

This is the pushdown stack storage, which contains all the variables and Data Descriptors associated with the program, including control words which indicate the dynamic status of the job as it is being executed.

THE B 5900 PROCESSOR STATE

Implemented on the B 5900 system is an operator set that is large system code compatible such that any object program that runs to correct termination on the B 6800 will do likewise on the B 5900. Since the BCL (Burroughs Common Language) character set and vector mode are not supported in the B 5900, object programs which deal with BCL or use vector mode are excluded.

STACK HISTORY AND ADDRESSING ENVIRONMENT

The addressing environment of the executing code stream consists of a set of local addressing spaces contained within stacks. These are called activation records (referred to as lexical regions elsewhere), and each consists of a set of variables addressed by an index relative to the base of the activation record.

Activation records are managed by use of two linked lists: the historical chain and the current lexical chain. History and lexical Links address the base word in the activation record.

The historical chain is a chronologically ordered list which consists of History Links connecting each activation record to the immediately preceding activation record. An historical chain pointer to the most recently initiated activation record is all that is required to access any activation record in the stack.

There may be inactive or partially initiated activation records in the stack. These are linked into the historical chain only, whereas other activation records, called entered activation records, are linked into both the historical and lexical chains.

The Lexical Link of an entered activation record points to the activation record which is its immediate global addressing space. Immediate global addressing space is defined in terms of the static program structure as follows: if B0 and B1 are blocks in the program, B0 contains B1, and activation records AR0 and AR1 correspond to B0 and B1, then AR0 is the immediate global addressing space of AR1.

The Lexical Link is the head of a lexical chain down to the most global addressing space of the activation record. That lexical chain defines the addressing environment associated with execution of the code stream bound to the activation record. The position of the activation record in its own lexical chain is its lexical level, where zero is the end of the chain (its most global level).

History links always point to an activation record in the same stack, but Lexical Links may point to an activation record in another stack. Therefore, an addressing environment may be mapped into a "cactus stack" structure.

The current addressing environment is the set of activation records addressed by the lexical chain whose head is the activation record bound to the executing code stream. It is called the topmost activation record and is the first entered activation record on the historical chain. The lexical level of the topmost activation record is noted LL, and there are LL+1 activation records in its environment. A lexical chain pointer to the topmost activation record is required for accessing the current environment.

A general reference to an item in the current environment takes the form of an (Lambda, Delta) address couple, where Lambda is a lexical level and Delta is an offset to the referenced item from the base of the activation record at level Lambda. Address couples are the means of addressing variable locations in the current environment by the executing code stream.

Processor management of the activation records in the stack utilizes the following logical registers:

F

The historical chain pointer to the most recent activation record in the stack. All activation records (inactive or entered) are accessible by following History Links from F.

LL

The lexical level in the current addressing environment of the topmost activation record – the level at which the processor is running.

D[LL]

The lexical chain pointer to the topmost activation record. Activation records in the current addressing environment are accessible by following Lexical Links from D[LL].

D[0]

A pointer to the most global activation record in the current addressing environment. System considerations require accessibility of the D[0] activation record.

Addressing is optimized by setting the remainder of the D registers from the D[LL] lexical chain, so that the definition of the D register array is:

D[i]

A pointer to the activation record at level i in the current addressing environment for i = LL down to 0, and LL <= 15.

Figure 3-5 is an example of an addressing environment example. The figure illustrates that the Lexical Link from the level 2 activation record is to another stack. Though it is not illustrated in the figure, it is possible that a cactus stack structure could be above the level 2 activation record.

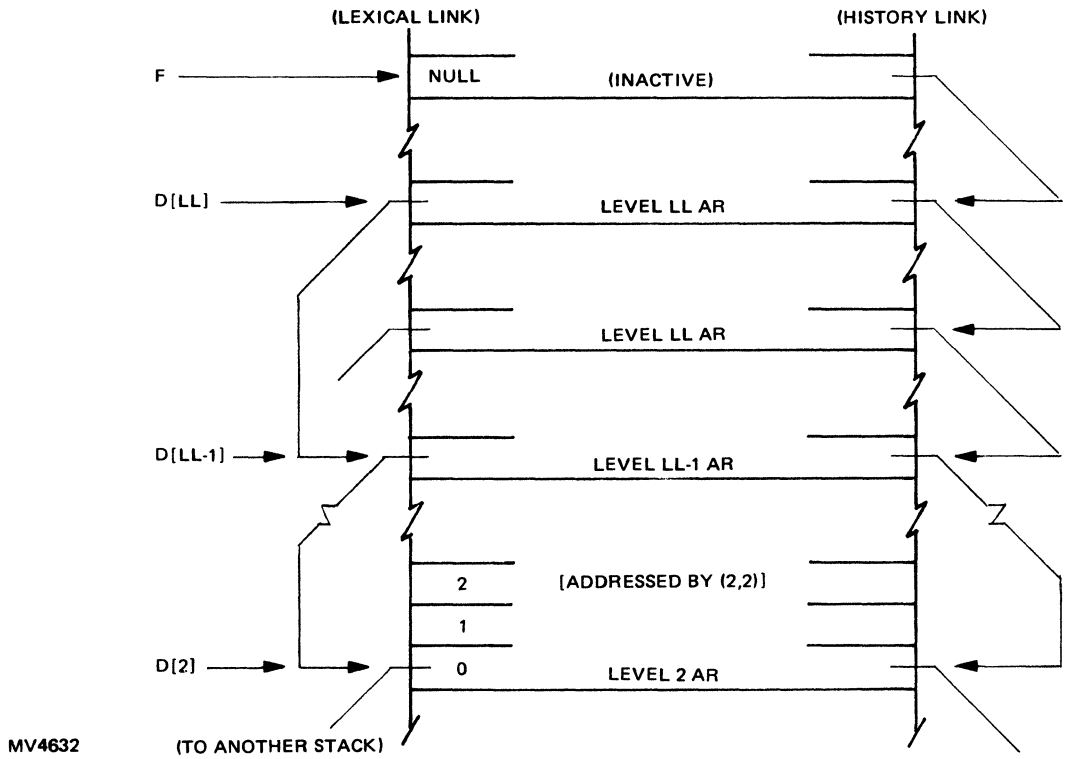


Figure 3-5. Addressing Environment Example

EXPRESSION STACK

Operator definition assumes an expression stack. Initial arguments are taken from and results are pushed onto this expression stack. The concepts of the expression stack and the current addressing environment are merged by treating the topmost activation record as the expression stack.

Variables local to the activation record are initialized by execution of operators which push items onto the expression stack. This "stack building code" is usually the first operator sequence executed following completion of entry into the activation record. At the conclusion of stack building code, the height of the expression stack is n words, where there are n local variable locations in the activation record.

The stack which contains the expression stack is identified by an integer value called Stack Number. The base and limit of the stack are obtained from the stack Descriptor. Activation records may occur in the stack below the topmost stack. The term, expression stack, describes that portion of the stack from the base of the topmost activation record to the limit of the stack.

Figure 3-6 shows a typical configuration of the topmost activation record after completion of stack building code and subsequent operator execution.

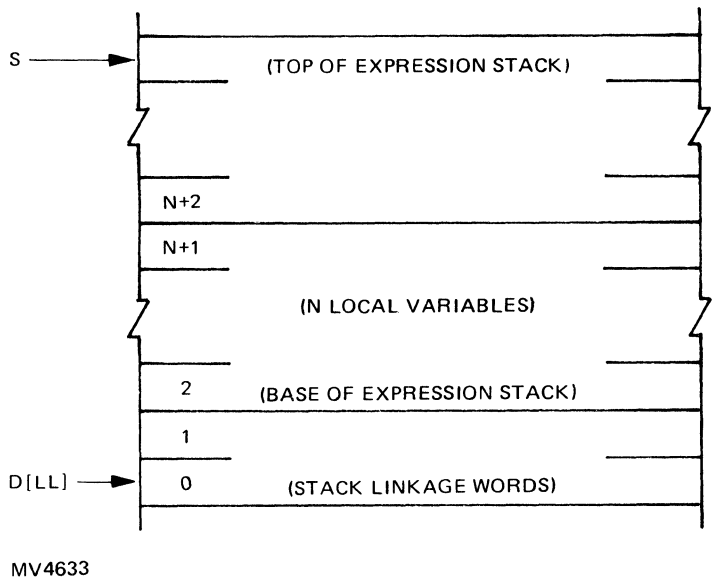


Figure 3-6. Topmost Activation Record Example

Processor management of the expression stack utilizes the following logical registers:

SNR
The stack number identification of the stack containing the expression stack.

S
The address of the top word in the expression stack.

Validity checking of stack accessing uses the following optimization registers:

BOSR
The base address of the stack containing the expression stack.

LOSR
The topmost (limit) address of the expression stack.

The B 5900 optimizes stack accesses by use of registers which hold the two top-of-stack items. The register contents are not addressable by way of an address couple, and are available only as dynamic arguments to operators.

EXECUTABLE CODE STREAMS

Variable length operator sequences are stored in arrays of program code words called code segments. Each program code word contains six 8-bit containers called syllables. (The mapping of operators into syllables will be specified in Section 4, Operator Set).

For each code segment there is a Descriptor which points to its memory location and specifies its length in words. Code Segment Descriptors for a program are collected in an array called a Segment Dictionary.

The term "code stream pointer" is used to describe a reference to the entry point of an operator sequence in a code segment. A code stream pointer consists of the following components: An address couple (SDLL, SDI) references the code segment Descriptor. SDLL is the Segment Dictionary lexical level. A user program Segment Dictionary is usually the level 1 activation record in its addressing environment, and the operating system Segment Dictionary is at level 0. SDI is the Segment Dictionary index to the code segment Descriptor relative to the base of the specified Segment Dictionary. The entry point in the code segment is indicated by PWI, the program word index relative to the base of the code segment, and PSI, the program syllable index of the first operator.

The processor code stream pointer consists of the following component registers:

SDLL (Segment Dictionary Lexical Level)
The lexical level at which the current Segment Dictionary is addressed.

SDI (Segment Dictionary Index)
The index in the Segment Dictionary to the current code segment Descriptor.

PWI (Program Word Index)
The index in the code segment to the code word containing the next operator.

PSI (Program Syllable Index)
The index in the code word to the next operator syllable.

Validity checking of code segment accessing uses the following optimization registers:

PBR (Program Base Register)
The memory address of the base of the current code segment.

PLR (Program Limit Register)
The memory address of the last word of the current code segment.

Figure 3-7 illustrates the processor code stream pointer.

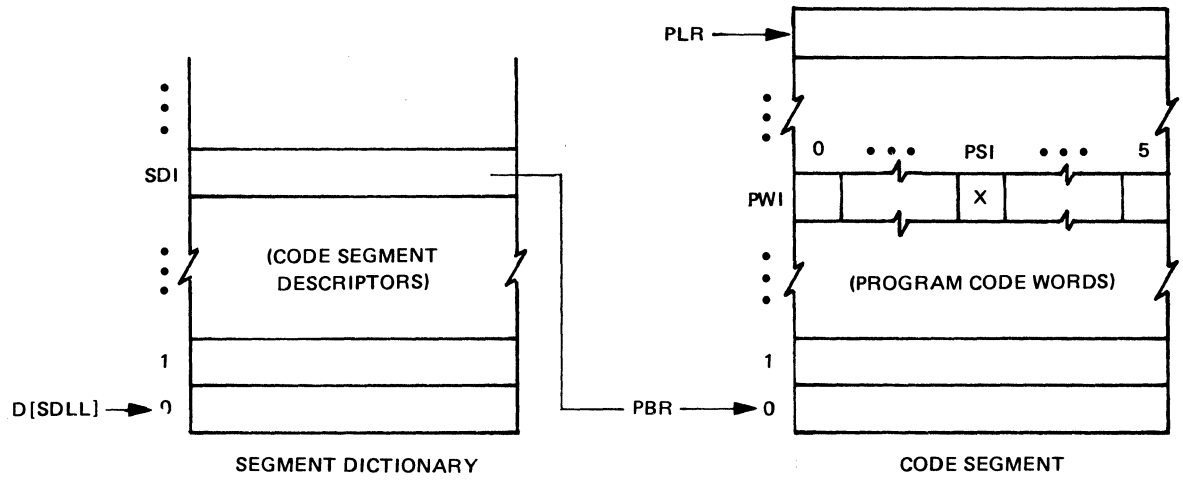


Figure 3-7. Processor Code Stream Pointer

Processor state also includes two Boolean attributes of the executing code stream and a global processor Halt Boolean:

CS (Control State)
: While CS is set, external interrupts are disabled. When it is reset (Normal State), they are enabled and may occur between operator executions.

RS (Restart State)
: If RS is set, the current operator was initiated at a restart state distinct from the normal initial state implied by RS being reset (Initial State).

HALT
: If HALT is enabled, processor execution will stop upon execution of a HALT (Conditional Processor Halt). If it is disabled, a HALT is treated as a NOOP (no operation).

GENERAL BOOLEAN ACCUMULATORS

Processor state includes four Boolean accumulators which are generally used by several operator groups. The use and definition of these four accumulators listed below will be discussed in Section 3, Operator Set.

TFFF
: The True False Flip-Flop.

OFFF
: The OverFlow Flip-Flop.

EXTF
: The External Sign Flip-Flop.

FLTF
: The Float Flip-Flop.

The Boolean TFOF (True False Occupied Flip-flop) is not required in B 5900. TFOF is used by Pointer compare operators to indicate that the operator has been restarted subsequent to a Paged array interrupt, but that the compare result was already determined. B 5900 provides a general operator restart mechanism using the RS Boolean. (See Executable Code Streams above.) This mechanism replaces the use of TFOF.

MISCELLANEOUS

Processor state includes the time of day clock and the processor identification:

TOD
The time of day clock, with values in 2.4 microsecond units.

PROC
The processor identification, composed of:

ERL
The Engineering Release Level.

SER NO
The system serial number.

ID
The processor identification number.

SECTION 4
B 5900 OPERATOR SET

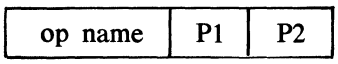
PRELIMINARY INFORMATION

Although operators are fetched from code segments, a code stream is considered to be a sequence of syllables fetched without regard to word boundaries. The two cases where word boundaries are relevant will be discussed separately with the operators LT48 (Insert 48-bit literal) and MPCW (make PCW).

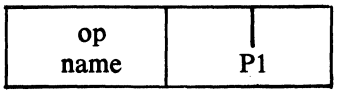
An operator is composed of an op code and up to four parameters. Op codes are generally one syllable and parameters, if any, are in the syllable following the op code. Parameter mapping into syllables varies, and explicit specifications appear in this section for operators requiring parameters. In this manual, the term "parameter" is to describe items in the code stream.

In those diagrams specifying op code and parameter interpretation, the operator name represents its op code value. Vertical bars (|) denote syllable boundaries, and colon (:) denotes parameter boundaries. Double vertical bars (||) denote word boundaries.

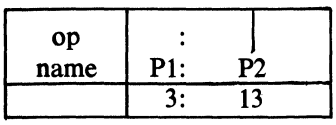
The following is an example diagram illustrating a 3-syllable operator, and includes 2 single-syllable parameters.



The next diagram shows a 3-syllable operator, two syllables of which are a single parameter.



The final diagram shows a 3-syllable operator including two parameters which are mapped into 2 syllables. P1 is 3 bits and P2 is 13 bits.



EXPRESSION STACK CONTROL

Most operators require stack items from the top of the expression stack; however, regardless of whether or not stack items are required, the operators leave their results on the top of the stack. Stack items required by operators, called arguments, are used and then deleted from the stack. To avoid excessive repetition, deletion of arguments is assumed for all operators, unless explicitly noted.

The expression stack utilizes the set of locations whose addresses are in the range {D(LL)+1 to LOSR}. The topmost activation record stack linkage word at D[LL] and D[LL]+1 is excluded. If the top of the stack is less than D[LL]+1 and an operator attempts to to use an expression stack argument, a Stack Underflow interrupt is generated. If an operator pushes a result onto the stack and at the conclusion of the push the top of the stack is equal to LOSR, a Stack Overflow interrupt is generated. This checking is assumed for all operators.

ARITHMETIC OPERATORS

Arithmetic operators require either one or two operands on top of the stack. If the items are not operands, an Invalid Stack Argument interrupt is generated.

Binary operators will generate a single-precision result if both operands are single-precision and a double-precision result if either or both operands are double-precision. Where required, single-precision is extended to double-precision prior to the operation by appending a second word of all zeros. The numeric value of the operand is not changed.

ADD (Add) P(80)

ADD requires two operands on top of the stack. The numeric values of the two operands are algebraically added and rounded, and the result is left on top of the stack. If the result is too large to be represented, an Exponent Overflow interrupt is generated.

SUBT (Subtract) P(81)

SUBT requires two operands on top of the stack. The numeric value of the top item is algebraically subtracted from the numeric value of the second item and rounded, and the result is left on top of the stack. If the result is too large to be represented, an Exponent Overflow interrupt is generated.

MULT (Multiply) P(82)

MULT requires two operands on top of the stack. The numeric values of the two operands are algebraically multiplied and rounded, and the result is left on top of the stack. If the result is too large to be represented, an Exponent Overflow is generated. If the result is too small to be represented, an Exponent Underflow interrupt is generated.

MULX (Extended Multiply) P(8F)

MULX requires two operands on top of the stack. Any single precision operand is extended to double-precision before the numeric values are algebraically multiplied and rounded. The double-precision result is left on top of the stack. If the result is too large to be represented, an Exponent Overflow interrupt is generated. If the result is too small to be represented, an Exponent Underflow interrupt is generated.

DIVD (Divide) P(83)

DIVD requires two operands on top of the stack. The numeric value of the second item is algebraically divided by the numeric value of the top item and rounded, and the result is left on top of the stack. If the divisor (top-of-stack operand) equals zero, a Divide by Zero interrupt is generated. If the result is too large to be represented, an Exponent Overflow interrupt is generated. If the result is too small to be represented, an Exponent Underflow interrupt is generated.

IDIV (Integer Divide) P(84)

IDIV requires two operands on top of the stack. The numeric value of the second item is algebraically divided by the numeric value of the top item. The fractional part of the floating point quotient is discarded, and the integer part is left on top of the stack represented as an integer.

If the divisor (top-of-stack operand) equals zero, a Divide by Zero interrupt is generated. If the result is too large to be represented as an integer, an Integer Overflow interrupt is generated.

RDIV (Remainder Divide) P(85)

RDIV requires two operands on top of the stack. The numeric value of the second item is divided by the numeric value of the top item. The integer quotient with remainder is generated but only the remainder is left on top of the stack. The sign of the result is the same as the sign of the second item (the dividend).

If the divisor (top-of-stack operand) equals zero, a Divide by Zero interrupt is generated. If the result is too large to represented as an integer, an Integer Overflow interrupt is generated.

CHSN (Change Sign) P(8E)

CHSN requires an operand on top of the stack. The sign of the operand is complemented and the result is left on top of the stack.

NORM (Normalize) V(8E)

NORM requires an operand on the top-of-stack; otherwise an Invalid Stack Argument interrupt is generated. If the operand is single precision, it is converted to normalized single-precision representation. If the operand is double-precision, it is converted to normalized double-precision representation.

If the top-of-stack is not an operand, an Invalid Stack Argument interrupt is generated. If the result is too small to be represented, an Exponent Underflow interrupt is generated.

RELATIONAL OPERATORS

The relational operators all require two operands on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The numeric value of the second item is algebraically compared to the numeric value of the top item, and a Boolean result is left on top of the stack. True is represented by a single-precision integer 1, and false is represented by a single-precision integer 0.

LESS (Less Than) P(88)

LESS leaves a true result if the second from top-of-stack operand is algebraically less than the top operand; otherwise LESS leaves a result of false.

LSEQ (Less Than Or Equal To) P(8B)

LSEQ leaves a true result if the second from top-of-stack operand is algebraically less than or equal to the top operand; otherwise LESQ leaves a result of false.

EQUL (Equal To) P(8C)

EQUL leaves a true result if the second from top-of-stack operand is algebraically equal to the top operand; otherwise a false results.

NEQL (Not Equal To) P(8D)

NEQL leaves a true result if the second from top-of-stack operand is algebraically not equal to the top operand; otherwise a false results.

GREQ (Greater Than Or Equal To) P(8A)

GREQ leaves a true result if the second from top-of-stack operand is algebraically greater than or equal to the top operand; otherwise a false result.

GRTR (Greater Than) P(89)

GRTR leaves a true result if the second from top-of-stack operand is algebraically greater than the top operand and a false result otherwise.

TYPE TRANSFER OPERATORS

The following type transformations may be invoked on the top-of-stack operand:

- Convert to single-precision integer using NTIA, NTGR.
- Convert to single-precision floating point using SNGT, SNGL.
- Convert to double-precision integer using NTGD.
- Convert to double-precision floating point using XTND. Additionally, a Data Descriptor may be converted to single-precision by SNGT or to double-precision by XTND.

NTIA (Integerize Truncated) P(86)

NTIA requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted to single precision integer representation by truncation, and the result is left on top of the stack. If the result is too large to be represented as an integer, an Integer Overflow interrupt is generated.

NTGR (Integerize Rounded) P(87)

NTGR requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted to single precision integer representation by rounding, and the result is left on top of the stack. If the result is too large to be represented as an integer, an Integer Overflow interrupt is generated.

SNGL (Set to Single-Precision Rounded) P(CD)

SNGL requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted to normalized single-precision representation by rounding and is left on top of the stack. If the result is too large to be represented in normalized single-precision form, an Exponent Overflow interrupt is generated. If the result is too small to be represented in normalized single-precision form, an Exponent Underflow interrupt is generated.

SNGT (Set to Single-Precision Truncated) P(CC)

SNGT requires an operand or Data Descriptor on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. If the top-of-stack item is an operand, it is converted to normalized single-precision representation by truncation and left on top of the stack. If the result is too large to be represented in normalized single-precision form, an Exponent Overflow interrupt is generated. If the result is too small to be represented in normalized single-precision form, an Exponent Underflow interrupt is generated.

If the top-of-stack item is a Data Descriptor, the element-size of the Descriptor must be single-precision or double-precision; otherwise an Invalid Stack Argument interrupt is generated. A single-precision Descriptor is left on the stack unchanged. The element-size of a double-precision Descriptor is set to single-precision, and if unindexed, the length field of the Descriptor is multiplied by 2. The modified Descriptor is left on top of the stack.

XTND (Set to Double-Precision) P(CE)

XTND requires an operand or a Data Descriptor on top of the stack; otherwise an Invalid Stack Argument interrupt is generated.

If the top-of-stack operand is double-precision, it is left on the stack unchanged. If the operand is single-precision, it is converted to double-precision representation by appending a second word whose fields are initialized to zero and by changing the tag of both words to 2; the double-precision result is left on the stack and its numeric value is unchanged.

If the top-of-stack item is a Data Descriptor, it must be single-precision or double-precision; otherwise an Invalid Stack Argument interrupt is generated. A double-precision Descriptor is left on the stack unchanged. The element-size of a single-precision Descriptor is set to double-precision, and when unindexed, the length field of the descriptor is divided by 2; any remainder is discarded. The modified Descriptor is left on top of the stack.

NTGD (Integerize Double-Precision Rounded) V(87)

NTGD requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted to double-precision integer representation, and the result is left on top of the stack. If the result is too large to be represented as a double-precision integer, an Integer Overflow interrupt is generated.

SCALING OPERATORS

The following are in the Scaling operator group.

Scale Left Operators

Scale left operators perform multiplication of an operand on top of the stack by ten raised to a power specified by a scale factor. The scale factor may be a dynamic argument or a static parameter.

The item to be scaled must be an operand; otherwise an Invalid Stack Argument interrupt is generated. An NTGD (integerize double-precision rounded) operation is performed if required, and if the operand cannot be integerized, an Integer Overflow interrupt is generated.

If the scale factor is a dynamic argument, it must be an operand; otherwise an Invalid Stack Argument interrupt is generated. It is integerized with rounding if required, and if it cannot be integerized, an Integer Overflow interrupt is generated. Both the static scale factor parameter and the integerized dynamic scale factor must be in the range (0 to 12); otherwise the original operand to be scaled is left on top of the stack, and OFFF (overflow flip-flop) is set to 1.

The result of the multiplication is left on top of the stack represented as a single-precision or double-precision integer, depending on its magnitude. The result is single-precision for the range (0 to 2^{38}), and double-precision for the range (2^{39} to 2^{77}). If it is greater than or equal to (2^{78}), an indeterminate double-precision integer is left on top of the stack, and OFFF (Overflow Flip-Flop) is set to 1.

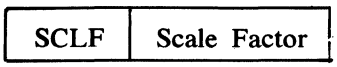
Scale left stack arguments (the operand to be scaled and the dynamic scale factor) are required to be operands.

The dynamic scale factor is checked to verify that the integer value is in the range (0 to 12), rather than the value modulo 16.

For all scale factor values, scale left operators correctly produce a single or double-precision integer result, depending on its magnitude.

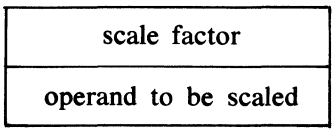
SCLF (Scale Left) P(C0)

The top-of-stack operand is multiplied by ten raised to the power specified by the scale factor. The resultant single-precision or double-precision integer is left on top of the stack. The scale factor is a parameter:



DSLF (Dynamic Scale Left) P(C1)

The operand to be scaled is multiplied by ten raised to the power specified by the scale factor. The resultant single-precision or double-precision integer is left on top of the stack. Both arguments are required on top of the stack:



Scale Right Operators

Scale right operators perform division of an operand on top of the stack by ten raised to a power specified by a scale factor. The scale factor may be a dynamic argument or a static parameter. The results of the division are the quotient represented as an integer, the remainder represented as a decimal (hex character) sequence, or a combination of the two.

The item to be scaled must be an operand; otherwise an Invalid Stack Argument interrupt is generated. An NTGD (integerize double-precision rounded) operation is performed if required, and if the operand cannot be integerized, an Integer Overflow interrupt is generated.

If the scale factor is a dynamic argument, it must be an operand; otherwise an Invalid Stack Argument interrupt is generated. It is integerized with rounding if required. If it cannot be integerized, an Integer Overflow interrupt is generated, and if the result is a valid integer but not in the range (0 to 12), an Invalid Argument Value interrupt is generated. If the scale factor is a parameter and it is not in the range (0 to 12), an Invalid Code Parameter interrupt is generated.

Scale right operators leave on top of the stack either the quotient of the division, the remainder, or both the quotient and remainder. The quotient is represented as a single-precision integer if its magnitude is in the range (0 to 2^{38}) and as a double-precision integer for the range (2^{39} to 2^{77}). The magnitude of the quotient cannot exceed 2^{77}. The remainder is a single-precision operand interpreted as a left-justified decimal (hex) sequence. The number of significant decimal digits is equal to the scale factor, and each digit is in the range (hex "0" to hex "9"). Scale right stack arguments (the operand to be scaled and the dynamic scale factor) are required to be operands.

The dynamic scale factor is checked to verify the integer value is in the range (0 to 12), rather than the value modulo 16.

For a scale factor of zero, scale right operators whose results include the quotient produce either a single-precision or a double-precision integer result, depending on the magnitude of the operand. Scale right operators with results that include the remainder produce a remainder of zero.

SCRS (Scale Right Save) P(C4)

SCRS leaves the quotient on top of the stack and the remainder is left second from the top of the stack. The operand to be scaled is required on top of the stack, and the scale factor is a parameter:

SCRS	Scale Factor

DSRS (Dynamic Scale Right Save) P(C5)

DSRS leaves the quotient on top of the stack and the remainder is left second from the top of the stack. The scale factor and the operand to be scaled are required on top of the stack:

scale factor
operand to be scaled

SCRT (Scale Right Truncate) P(C2)

Only the quotient is left on top of the stack. The operand to be scaled is required on the stack, and the scale factor is a parameter:

SCRT	Scale Factor

DSRT (Dynamic Scale Right Truncate) P(C3)

Only the quotient is left on top of the stack. The scale factor and the operand to be scaled are required on top of the stack:

scale factor
operand to be scaled

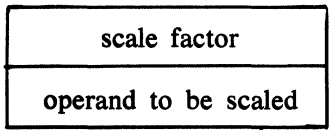
SCRR (Scale Right Rounded) P(C8)

If the most significant digit of the remainder is greater than or equal to five, the magnitude of the quotient is rounded up; that is, if the quotient is 1, it rounds to 2. If the quotient is –1, it rounds to –2. Only the quotient is left on top of the stack. The operand to be scaled is required on top of the stack, and the scale factor is a parameter:

SCRR	Scale Factor

DSRR (Dynamic Scale Right Rounded) P(89)

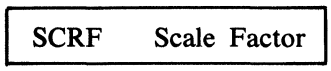
If the most significant digit of the remainder is greater than or equal to five, the quotient is rounded up, and only the quotient is left on top of the stack. The scale factor and the operand to be scaled are required on top of the stack:



SCRF (Scale Right Final) P(C6)

Only the remainder is left on top of the stack. EXTF (External Sign Flip-Flop) is set to 1 if the mantissa sign of the operand to be scaled is negative; otherwise the EXTF is 0. OFFF (overflow flip-flop) is set to 1 if the quotient is any non-zero value; otherwise the OFFF is 0.

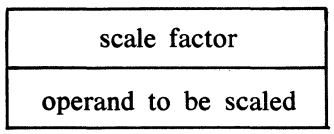
The operand to be scaled is required on top of the stack, and the scale factor is a parameter:



DSRF (Dynamic Scale Right Final) P(C7)

Only the remainder is left on top of the stack. EXTF (External Sign Flip-Flop) is set to 1 if the mantissa sign of the operand to be scaled is minus; otherwise EXTF is 0. OFFF (Overflow Flip-flop) is set to 1 if the quotient is any non-zero value; otherwise OFFF is 0.

The scale factor and the operand to be scaled are required on top of the stack:



LOGICAL OPERATORS

Logical operators require one or two top-of-stack items. They may be of any type. The items are interpreted as 48-bit vectors unless one or both are double-precision operands. Then they are interpreted as 96-bit vectors, and if only one of two items is double-precision, the other is extended with 48 0 bits.

The logical operation is applied in parallel to each bit of the vectors, and the result is left on top of the stack. The tag of the result is determined as follows: If the operation is applied to double-precision items, the result is double-precision. Otherwise, the tag of the result is the tag of the top-of-stack item for the unary operator LNOT and the tag of the second-from-top item for the binary operators.

LNOT (Logical Not) P(92)

LNOT requires a single top-of-stack item. All bits of the vector are complemented, and its tag remains unchanged.

LAND (Logical And) P(90)

LAND requires two top-of-stack items. The logical AND of the two bit vectors is left on top of the stack.

LOR (Logical Or) P(91)

LOR requires two top-of-stack items. The logical OR of the two bit vectors is left on top of the stack.

LEQV (Logical Equivalence) P(93)

LEQV requires two top-of-stack items. The logical EQV (equivalence) of the two bit vectors is left on top of the stack.

RELATIONAL OPERATOR

SAME (Logical Equality) P(94)

SAME requires two top-of-stack items. If all corresponding bits of the two items (including tag bits) have the same value, a true result is left on top of the stack; otherwise a false result is left. If both items are double-precision, the bit vector interpretation includes the second words. Note that if only one item is double-precision, the result is necessarily false.

True and false are represented in the same way as in arithmetic relational operators. True is a single-precision integer 1, and false is a single-precision integer 0.

LITERAL OPERATORS

Literal operators place a single-precision constant on top of the stack. They do not use any initial top-of-stack items.

ZERO (Insert Literal Zero) P(B0)

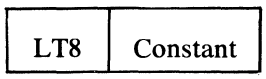
ZERO leaves on top of the stack a single-precision word with all bits initialized to zero.

ONE (Insert Literal One) P(B1)

ONE leaves on top of the stack a single-precision word with bit zero set to ONE and all other bits initialized to ZERO.

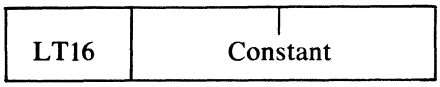
LT8 (Insert 8-Bit Literal) P(B2)

LT8 leaves on top of the stack a single-precision word with the field [7:8] set from its one syllable parameter and all other bits initialized to ZERO.



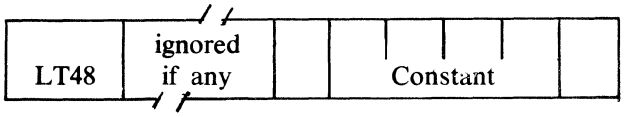
LT16 (Insert 16-Bit Literal) P(B3)

LT16 leaves on top of the stack a single-precision word with the field [15:16] set from its two-syllable parameter and all other bits initialized to ZERO.



LT48 (Insert 48-Bit Literal) P(BE)

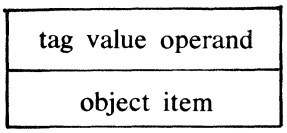
LT48 leaves on top of the stack a single-precision word whose 48 bits are set from its six-syllable parameter. The parameter is taken from the first code word following the LT48 op code. "Padding" syllables from the op code to the end of the word containing the op code are ignored when they occur.



TYPE TRANSFER OPERATORS

STAG (Set Tag) V(B4)

STAG requires a tag value and an object item on top of the stack, leaving as the result an item whose tag is the tag value and whose 48 bits are copied from the object item.

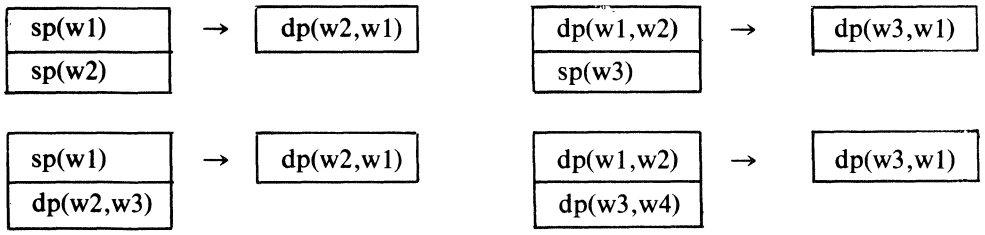


The tag value must indicate an operand; otherwise an Invalid Stack Argument interrupt is generated. The tag of the object item is set from field [3:4] of the first word of the tag value operand. There is no restriction on the initial type of the object item.

JOIN (Set Two Singles to Double) V(42)

JOIN requires two operands on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. A double-precision item is constructed from the two operands, and the result is left on top of the stack.

The first and second words of the double-precision result are taken from the first words of the second and top operands respectively. The following possibilities arise from combinations of single-precision and double-precision operands:



SPLT (Set Double to Two Singles) V(43)

SPLT requires an operand on top of the stack; otherwise, an Invalid Stack Argument interrupt is generated. Two single-precision items are constructed from the operand and left on top of the stack.

If the operand is single-precision, it is left on the stack and a single precision integer 0 is pushed on the stack above it. If the operand is double-precision, the two words of the operand are converted to two single-precision items. The first word is pushed on the stack first, and the second word is left on top of the stack.

EVALUATE WORD STRUCTURE OPERATORS

RTAG (Read Tag) V(B5)

RTAG requires one item on top of the stack, and the result is a single-precision integer whose value is the tag of the item.

CBON (Count Binary Ones) V(BB)

CBON requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The number of bits in the operand which have the value 1 are counted. If the operand is double-precision, all 96 bits are examined. On top of the stack, CBON leaves a single-precision integer whose value is the 1s count.

LOG2 (Leading One Test) V(8B)

LOG2 requires one item on top of the stack and leaves in place of it a single-precision integer indicating that the leading bit in the item has the value 1. If all bits in the item are 0, LOG2 leaves an integer 0; otherwise, the integer value is the bit number plus one of the highest-order 1-bit. Only the first word of a double-precision operand is examined.

WORD MANIPULATION OPERATORS

Word manipulation operators provide the capability to alter any "partial field" of a word in the stack, called the destination, in cases based on a field of another word in the stack called the source. The following operations are provided:

1. Set a single destination bit,
2. Reset a single destination bit,
3. Create a destination whose low-order field is set from a field of the source,
4. Set a field of the destination from the low-order field of the source operand,
5. Set a field of the destination from a field of the source.

Destination and source items may be of any type, and the altered destination item is left on top of the stack. If the source is a double-precision item, the field is taken from its first word, and the second word is discarded. If the destination is a double-precision item, the bit or field altered is in its first word, and the second word is retained unchanged in the double-precision result. The following terms are used for bit or field specifications:

Db
: The destination bit to be set or reset, or the high-order bit of the destination field.

Sb
: The high-order bit of the source field.

Len
: The length of both the source and destination fields.

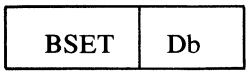
There are static and dynamic operators corresponding to each of the five operations. The static operators take Db, Sb, and Len specifications from parameters as required, and the dynamic operators take them from top-of-stack operands.

If dynamic Db, Sb, and Len specification items are not operands, an Invalid Stack Argument interrupt is generated. These items are integerized with rounding if integerizing is required, and if they cannot be integerized, an Integer Overflow interrupt is generated. All Db and Sb values must be in the range (0 to 47), and Len values must be in the range (0 to 48). Static operators generate an Invalid Code Parameter interrupt if any of these values are invalid, and dynamic operators generate an Invalid Argument Value interrupt if any values are invalid.

The effect of word manipulation operators are shown as an assignment to a field of the destination word. The remainder of the destination word is not changed. Note that Len = 0 is a valid specification of a null field; in this case the destination will not be altered at all.

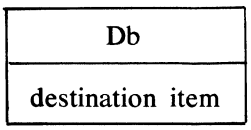
BSET (Bit Set) P(96)

BSET sets the following single-destination bit: destination [Db:1] := 1. The destination is the only required top of stack item, and Db is specified by a parameter:



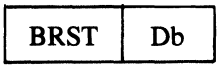
DBST (Dynamic Bit Set) P(97)

DBST sets the following single-destination bit: destination.[Db:1] := 1. The required initial stack state includes Db:



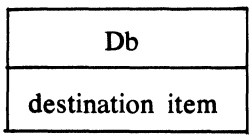
BRST (Bit Reset) P(9E)

BRST resets the following single-destination bit: destination.[Db:1] := 0. The destination is the only required top of stack item, and Db is specified by a parameter:



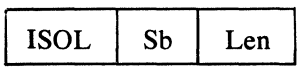
DBRS (Dynamic Bit Reset) P(9F)

DBRS resets the following single-destination bit: destination [Db:1] := 0. The required initial stack state includes Db:



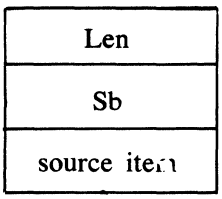
ISOL (Field Isolate) P(9A)

ISOL creates a single-precision destination word initialized to 0, and then sets the low-order field of the word from a field of the source in the following manner: destination := 0; destination.[Len-1:Len] := source.[-Sb:Len]. The source is the only required top of stack item, and Sb and Len are specified by parameters:



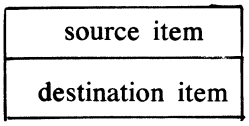
DISO (Dynamic Field Isolate) P(9B)

DISO creates a single-precision destination word initialized to 0, and then sets the low-order field of the word from a field of the source in the following manner: destination := 0; destination.[Len-1:Len] := source.[-Sb:Len]. The required initial stack state includes Len and Sb:

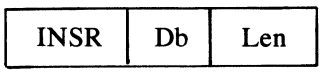


INSR (Field Insert) P(9C)

INSR sets a field of the destination from the low-order field of the source in the following manner: destination [Db:Len] := source [Len-1:Len]. The required initial stack state includes only the source and destination:

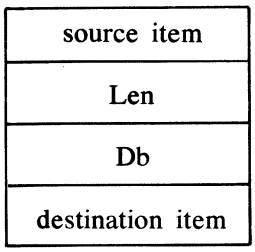


Values for Db and Len are specified by parameters:



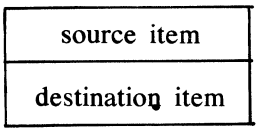
DINS (Dynamic Field Insert) P(9D)

DINS sets a field of the destination from the low-order field of the source in the following manner: destination [Db:Len] := source [Len-1:Len]. The required initial stack state includes Len and Db (except for DINS, where the source item is required on top of the stack):

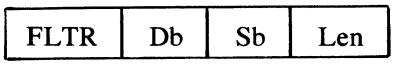


FLTR (Field Transfer) P(98)

FLTR sets a field of the destination from a field of the source in the following manner: destination [Db:Len] := source [Sb:Len]. The required initial stack state includes only the source and destination:

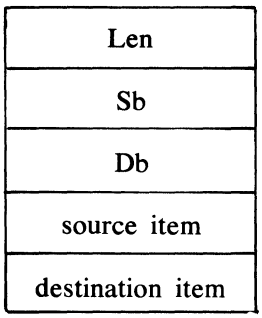


Values for Db, Sb, and Len are specified by parameters:



DFTR (Dynamic Field Transfer) P(99)

DFTR sets a field of the destination from a field of the source in the following manner: destination [Db:Len] := source [Sb:Len]. The required initial stack state includes Len, Sb and Db:

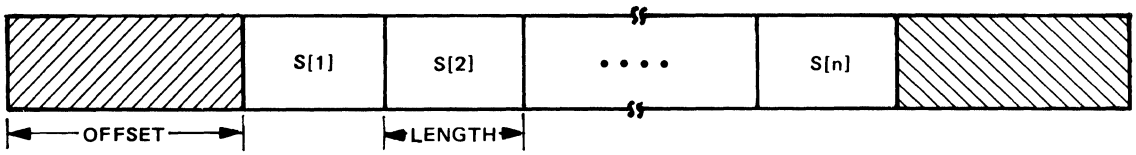


SPECIAL INTERPRETATIONS

The following paragraphs define special interpretations applied to certain operators.

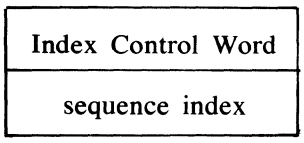
OCRX (Occurs Index) V(85)

OCRX is intended to aid COBOL in generating indices to sequences within a linear record structure. For example, assume that a record R contains a sequence S with elements s[1] to s[n]:



The index for s[i] relative to the base of R can be computed by the following function: Relative Index (offset, length, i) = offset + (i–1) *length; where i is in the range (1 to n).

OCRX leaves on top of the stack the result of the Relative Index function applied to values derived from two top-of-stack arguments:



The Index Control Word (ICW) must be a single-precision word. The ICW contains the argument values offset and length as well as the sequence size (n in the example). ICW interpretation is:

ICW	Sequence	ICW
Length	Size	Offset
[47:16]	[31:16]	[15:16]

ICW length [47:16]
The length of an element in the sequence

Sequence size [31:16]
The number of elements in the sequence

ICW offset [15:16]
The offset from the base of the record to the start of the sequence

The sequence index (i in the example) is integerized with rounding when required. If the ICW is not a single-precision word or if the sequence index is not an operand, an Invalid Stack Argument interrupt is generated. If the sequence index cannot be integerized, an Integer Overflow interrupt is generated.

OCRX first checks for valid application of the Relative Index function by confirming that sequence index is in the range (1 to sequence size). If it is not, an Invalid Index interrupt is generated. Otherwise, OCRX leaves a single-precision integer on top of the stack. The value of this integer is as follows: Relative Index (ICW offset, ICW length, sequence index).

REFERENCE GENERATION AND EVALUATION OPERATORS

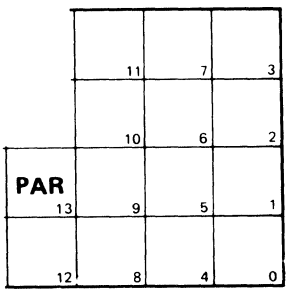
The primary concern of this operator group is the generation and evaluation of indirect references and chains of indirect references. The group consists of reference generation operators (generators) and two kinds of operators that evaluate references either to read a target item onto the stack (read evaluators) or to store an item from the stack into a target location (store evaluators).

Evaluation of Indirect References

Read and store evaluators share the general capability of processing a chain of indirect references in order to locate a target item. Reference chains may be composed of address couple parameters, IRWs (NIRWs and SIRWs), Indexed Word DDs and PCWs.

The definition of valid target items and allowable reference chains is dependent upon the function of the particular operator, but the evaluation of each element of a reference chain and of IRW chains is common to the operator group. The following sections define the evaluation of each reference form, the IRW chain evaluation, and the notation used for each operator to specify allowable reference chains and valid target items.

ADDRESS COUPLE PARAMETERS



An address couple parameter is interpreted identically to the address couple component of an NIRW. Evaluation of the parameter consists of reading the item in the stack addressed by (Lambda, Delta).

When an address couple is evaluated, Lambda must be less than or equal to LL, and for Lambda=LL, the address of the referenced stack location must be less than or equal to the address of the top-of-stack; otherwise, an Invalid Reference interrupt is generated. Note that the result of address couple evaluation may vary according to the current addressing environment.

NIRWS

Evaluation of an NIRW consists of replacing it by the item in the stack referenced by the NIRW address couple.

When an NIRW is evaluated, Lambda must be less than or equal to LL, and for Lambda=LL, the address of the referenced stack location must be less than or equal to the address of the top-of-stack; otherwise, an Invalid Reference interrupt is generated. Note that the result of NIRW evaluation may vary according to the current addressing environment.

SIRWS

Evaluation of an SIRW consists of replacing it by the item contained in the memory location referenced by the SIRW. Presence Bit interrupts are generated if either the stack vector array or the referenced stack is absent.

The result of evaluation of an SIRW is constant regardless of the current addressing environment. No validity check is performed during SIRW evaluation. SIRWs are created from NIRWs, and the NIRW components are verified at that time.

INDEXED WORD DDS

The evaluation of a Data Descriptor as an indirect reference is possible only if it is an Indexed Word DD. In that case, the Indexed Word DD is replaced by the item contained in the memory location that the Indexed Word DD addresses.

If an operator (for example, NXLV) has to read a word referenced by an Indexed Word DD and if the Indexed Word DD is present, the word it references can be accessed. If the Indexed Word DD is absent, the associated mom DD is read; if the mom DD is absent, a Presence Bit interrupt is generated; if the mom DD is present, the referenced word can be read.

In cases where Indexed Word DD evaluation produces an operand, the element type of the operand is determined by the element-size field of the Indexed Word DD (the last if a sequence of Indexed Word DDs was evaluated). The element-size value of single-precision or double-precision overrides the tag of the target operand. All operators which evaluate Indexed Word DDs obey this convention.

PCWS

In the context of reference chain evaluation, evaluation of a PCW consists of an accidental entry. The PCW is assumed to point to a function with no parameters, whose returned value will be either the target of the chain or another valid reference. The net result of the accidental entry is that the PCW is replaced by the item the PCW function returns.

Evaluation of the PCW requires an operator-generated entry into the environment and code stream referenced by the PCW. Under normal conditions, the function will terminate with a RETN (return) operator which leaves an item on top of the stack. If it does not, the item on top of the stack is used incorrectly, as if it were another result.

As a result of accidental entry and subsequent return, the processor executes the operator which initiated the accidental entry. The operator will normally restart from its initial state, and the accidental entry result will be a valid initial state for the operator. VALC (Value Call) is a notable exception; its restart state is unique from its initial state.

IRW CHAINS

The term IRW is used for a reference which is either an NIRW or an SIRW. Throughout this group of operators, those which evaluate multiple references will evaluate a sequence of one or more IRWs, called an IRW chain, wherever a single IRW may be evaluated.

Evaluation of the IRW chain consists of successive IRW evaluations, starting with the head of the chain, until IRW evaluation does not produce an IRW. The net result is that the head of the chain is replaced by the non-IRW result.

The only restriction on IRW chain evaluation is that once an SIRW is evaluated, an NIRW may not be evaluated. Thus, a valid IRW chain may be an NIRW chain, an SIRW chain, or an NIRW chain --> SIRW chain (an NIRW chain whose result is the head of an SIRW chain). If in IRW chain evaluation SIRW evaluation produces an NIRW, an Invalid Reference Chain interrupt is generated.

REFERENCE CHAINS

Operators which evaluate reference chains start from an initial reference and apply successive reference evaluation, according to a set of chaining rules, until a target item is produced. For each such operator, the set of initial references and targets will be specified using the following notation:

<Initial Reference>
    ::= (set of reference items)

<targets>
    ::= (set of target items)

Chaining rules are specified by showing the valid evaluation results for each reference form which may be a part of the chain. The form of such specification is:

    reference --> (evaluation results)

In this specification, where "-->" indicates an evaluation of the reference as defined in the preceding sections. Evaluation results can include reference forms or an Initial Reference, any of which is subsequently evaluated, or a target. Evaluation of the chain will continue until a target is encountered or until reference evaluation produces an item which is not a valid result. If chain evaluation terminates with an invalid item, an Invalid Reference Chain interrupt is generated.

Chaining rule notation is illustrated by the following example:

<Initial Reference>
    ::= (IRW Chain, Indexed Word DD)

<target>
    ::= (Operand)

<IRW Chain>
    --> (Indexed Word DD); PCW; <target>

(Indexed Word DD)
    --> (Indexed Word DD); <target>

PCW
    --> <Initial Reference>

REFERENCE GENERATION OPERATORS

NAMC (Name Call) P(40)-P(7F)

NAMC is a literal operator for transforming an address couple in the code stream into an NIRW.

An NIRW is created, the address couple is copied into it, and the NIRW is left on top of the stack. Since NAMC does not evaluate the address couple, the Lambda component is not validated.

STFF (Stuff) P(AF)

STFF converts the NIRW on top of the stack into an SIRW. If STFF encounters an SIRW on top of the stack it terminates leaving the SIRW. If the top-of-stack item is not an NIRW or SIRW, an Invalid Stack Argument interrupt is generated.

If Lambda>LL, or Lambda=LL and address(stack location) > address(top-of-stack), an Invalid Reference interrupt is generated. Otherwise, the SIRW is constructed to point to the word in the stack addressed by (Lambda, Delta). The unused fields are reset to 0.

INDX (Index) P(A6)

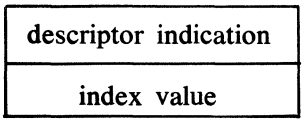
INDX applies an integer index to a Data Descriptor (DD) and leaves on top of the stack an Indexed DD pointing to the specified element. If the DD is a Word DD, the result will be an Indexed Word DD, and if it is a string DD, the result will be a Pointer. The DD may be a copy DD on the stack initially, or an IRW chain may address the DD (mom or copy):

<Descriptor Indication>
    ::= (IRW chain, copy DD (target))

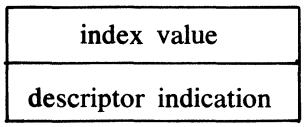
<target>
    ::= (mom DD, copy DD)

IRW chain
    --> <target>

INDX requires the descriptor indication and an operand index value on top of the stack in either order:



OR



If the index value is not an operand or the Descriptor Indication is not a copy DD (of a valid element-size) or the head of an IRW chain, an Invalid Stack Argument interrupt is generated. If an IRW chain does not produce a DD (of a valid element-size), an Invalid Reference Chain interrupt is generated. The index value is integerized with rounding if required. If it cannot be integerized, an Integer Overflow interrupt is generated, and if the result is a valid integer but not in the range (0 to DD.length–1), an Invalid Index interrupt is generated. (Both DD length and the index value are assumed to be in DD element-size units).

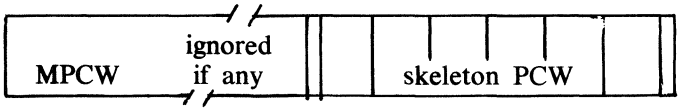
An Indexed Word DD or a Pointer is constructed according to the value of DD element-size. If it is single-precision or double-precision, an Indexed Word DD is constructed with the index value copied into its index field. If the DD has a double-precision element-size, the index is doubled before being copied into the index field. If the DD is EBCDIC or hex, a Pointer is constructed with its word index and char index fields computed to reference the sub-word element. If the computed word index exceeds 2^{15}, an Invalid Index interrupt is generated.

The Indexed DD is marked as a copy. If the data segment is present, the address field is set to point to the base address of the segment. If it is absent, the address field is set to the address of the mom Descriptor. Present, read only, and element-size fields are copied into the Indexed DD from the original DD.

If the data segment is paged, INDX resolves the paging by performing another level of indexing. If the DD is a Word DD, the referenced page number is (index DIV 256) and if the DD is a string DD, the referenced page number is (word index DIV 256). The specified page Descriptor is fetched, and its index field or its word index and char index fields are set as follows to reference the specified element relative to the base of the page. If the DD is a Word DD, the index field is set to (index MOD 256), and if the DD is a string DD, the word index field is set to (word index MOD 256) and the char index field is unchanged. The read only bit and the element-size field of the <target> paged Descriptor are copied into the page Descriptor, the copy and indexed bits are set, and the paged bit is reset. The presence bit and the address field of the page Descriptor remain in the resulting Indexed DD, with the address field modified to reference the associated mom page Descriptor if the page Descriptor was an absent mom DD. INDX will never generate an Indexed DD with the paged bit set.

MPCW (Make PCW) P(BF)

MPCW is a literal operator which constructs a PCW at the top of the stack from a 6-syllable parameter in the code stream. The parameter is taken from the first code word following the MPCW op code. "Padding" syllables from the op code to the end of the word containing the op code are ignored.



The parameter is assumed to be a valid PCW skeleton. It is inserted at the top of the stack and tagged as a PCW. Stack number, the only dynamic PCW field, is copied into the PCW from SNR.

READ EVALUATION OPERATORS

VALC (Value Call) P(00)-P(3F)

VALC evaluates a reference chain whose head is an address couple in order to locate a target operand which is left on top of the stack.

Reference chain evaluation performed by VALC is:

<Initial reference	::=	(address couple)
<target>	::=	(operand)
(address couple)	-->	IRW chain Indexed Word DD Word DD <target>
IRW chain	-->	Indexed Word DD Word DD PCW <target>
Indexed Word DD	-->	Indexed Word DD Word DD <target>
Word DD	-->	Indexed Word DD (see discussion below)
PCW	-->	IRW chain Indexed Word DD Word DD <target>

If reference evaluation produces an item which is not a valid result according to the above chain evaluation rules, an Invalid Reference Chain interrupt is generated.

If reference evaluation produces a Word DD, VALC will attempt an INDX (index) operation on the Word DD using the top item on the stack as an index value. If that item is not an operand, an Invalid Stack Argument interrupt is generated. If the indexing is successful, the resultant Indexed Word DD is evaluated (see INDX for further specification and possible error conditions).

The result of VALC is a single-precision or double-precision operand which is left on top of the stack. The target is placed directly on the stack except in the following cases where type conversion is performed before the modified target is left on the stack:

1. If one or more Indexed Word DDs are evaluated, the element-size of the last Indexed Word DD specifies single-precision, and the target is a double-precision operand, the first addressed target word is placed on the stack with a tag of 0.
2. If one or more Indexed Word DDs are evaluated, the element-size of the last Indexed Word DD specifies double-precision, and the target is a single-precision operand, the addressed target and its successor from memory are placed on the stack with a tag of 2.

NXLV (Index and Load Value) P(AD)

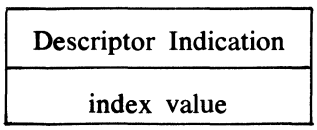
NXLV performs an INDX (index) operation to produce an Indexed Word DD and then evaluates the Indexed Word DD. The result must be an operand; otherwise, an Invalid Reference Chain interrupt is generated. The operand is treated exactly as is the VALC target. It is placed directly on top of the stack except in two cases where type conversion is performed before the modified target is left on the stack. (See preceding VALC target discussion).

The required initial stack state is the same as that for INDX except that the DD must be a Word DD:

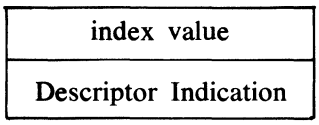
Descriptor Indication
    ::= {IRW chain, copy Word DD (target)}

<target>
    ::= (mom Word DD, copy Word DD)

IRW chain
    --> <target>



OR



If the index value is not an operand or the Descriptor Indication is not a copy Word DD or the head of an IRW chain, an Invalid Stack Argument interrupt is generated. If an IRW chain does not produce a Word DD, an Invalid Reference Chain interrupt is generated. If the index value cannot be integerized, an Integer Overflow interrupt is generated. If the result is a valid integer but not in the range (0 to DD length–1), an Invalid Index interrupt is generated.

NXLN (Index and Load Name) P(A5)

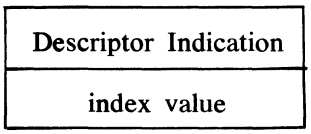
NXLN performs an INDX (index) operation to produce an Indexed Word DD and then evaluates the Indexed Word DD. The result must be a DD (whose element-size may be of any defined value); otherwise, an Invalid Reference Chain interrupt is generated. The DD is marked as a copy. If it is an absent mom DD, its address field is set to the memory address of the mom. The DD is left on top of the stack.

The required initial stack state is the same as that for INDX except that the DD must be a Word DD as shown in the following example:

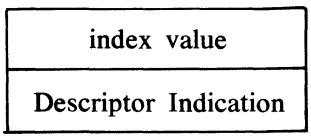
Descriptor Indication
    ::= (IRW chain, copy Word DD (target))

<target>
    ::= (mom Word DD, copy Word DD)

IRW chain
    --> <target>



OR



If the index value is not an operand or the Descriptor Indication is not a copy Word DD or the head of an IRW chain, an Invalid Stack Argument interrupt is generated. If an IRW chain does not produce a Word DD, an Invalid Reference Chain interrupt is generated. If the index value cannot be integerized, an Integer Overflow interrupt is generated. If the result is a valid integer but not in the range (0 to DD length–1), an Invalid Index interrupt is generated.

EVAL (Evaluate) P(AC)

The purpose of EVAL is to evaluate an indirect reference chain in order to locate a specific target and then leave the reference, whose evaluation produced the target, on top of the stack.

Reference chain evaluation performed by EVAL is:

<Initial Reference>
    ::= (IRW chain, Indexed Word DD)

<target>
    ::= (Operand, DD, Pointer)

IRW chain
    ::= (Indexed Word DD); PCW; <target>

Indexed Word DD
    --> (No evaluation – See below)

PCW
    --> <Initial Reference>

If a target is located, the reference whose evaluation produced the target is left on top of the stack as the result. If an Indexed Word DD is encountered, it is left as the result without being evaluated. In effect, Indexed Word DDs are treated as if they had been evaluated and a target had been the result.

If reference evaluation produces an item which is not a valid result according to the above chain evaluation rules, an Invalid Reference Chain interrupt is generated.

LOAD (Load) P(BD)

LOAD performs a single evaluation of the Initial Reference, and if the result is a target, it is left on top of the stack:

<Initial reference>
    ::= (IRW, Indexed Word DD)

<target>
    ::= (Operand, tag 4 word, Uninitialized Operand, IRW, any Data Descriptor)

IRW
    --> <target>

Indexed Word DD
    --> <target>

If the item on top of the stack is not an Initial Reference, an Invalid Stack Argument interrupt is generated, and if the reference evaluation result is not a target (it is a PCW or tag 3 word), an Invalid Reference Chain interrupt is generated. If the Initial Reference is a double-precision Indexed Word DD and the target is not an operand, an Invalid Reference Chain interrupt is generated.

If the Initial Reference is an IRW referencing a word with a double-precision tag or if the Initial Reference is a double-precision Indexed Word DD, both the addressed target word and its successor from memory are fetched. If the successor word has an odd tag, a Memory Protect interrupt is generated; otherwise, the two words are left on top of the stack as a double-precision operand.

If the Initial Reference is a single-precision Indexed Word DD addressing a double-precision operand, the addressed target word is left on top of the stack with a tag of 0.

If the Initial Reference is an IRW or a single-precision Indexed Word DD addressing a mom DD, the DD is left on top of the stack marked as a copy. When it is absent, the address field is set to the memory address of the mom DD.

In all other cases, the addressed target word is left on top of the stack without conversion.

LODT (Load Transparent) V(BC)

LODT performs a single evaluation of the Initial Reference and leaves the result on top of the stack, with no restriction placed on the type of the result, as shown in the following example:

```
<Initial reference>
     ::= (IRW, Indexed Word DD, Operand)

<target>
     ::= (any item)

IRW
     --> <target>

Indexed Word DD
     --> <target>
```

If the Initial Reference is an IRW or an Indexed Word DD, it is evaluated normally. When the Initial Reference is an operand, the operand is integerized with rounding if required, and the result is interpreted as a memory address from which the target is fetched.

If the item on top of the stack is not an Initial Reference, an Invalid Stack Argument interrupt is generated. If an operand Initial Reference cannot be integerized, an Integer Overflow interrupt is generated, and if the result is a valid integer but not in the range (0 to 2^{19}), an Invalid Argument Value interrupt is generated.

If the Initial Reference is an IRW or an operand addressing a word with a double-precision tag, or if the Initial Reference is a double-precision Indexed Word DD, LODT will leave the addressed target word and its successor from memory on top of the stack as a double-precision operand. If the Initial Reference is a single-precision Indexed Word DD addressing a double-precision operand, the addressed target word is left on top of the stack with a tag of 2 and a second word of 0. In all other cases, the addressed target word is left on top of the stack without conversion.

LODT restricts the Initial Reference to an IRW, an Indexed Word DD, or an operand, which is integerized and confirmed to be in the range (0 to 2^{19}) before being used as an address.

LODT obeys the normal convention that Indexed Word DD element-size overrides the tag of a target operand.

STORE EVALUATION OPERATORS

Normal Store Operators

Normal store operators evaluate a reference chain in order to store some item from the stack (the store object) into a Data Word target location. The store object may be of any type. Reference chain evaluation performed by store operators is:

<Initial reference>
    ::= (IRW chain, Indexed Word DD)

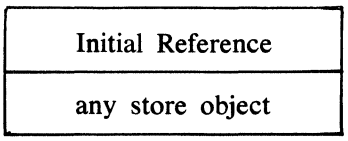
<target>
    ::= (Operand, tag 4 word, Uninitialized Operand)

IRW chain
    --> Indexed Word DD; PCW; <target>

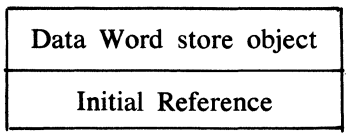
Indexed Word DD
    --> Indexed Word DD; <target>

PCW
    --> <Initial Reference>

The Initial Reference and the store object are required on top of the stack. Although, the Initial Reference is assumed to be the top item and the store object the second item, if the top item is a Data Word (even tag), the second item is assumed to be the Initial Reference. Note that a Data Word store object and the Initial Reference may be in either order, but if the store object has an odd tag, the Initial Reference must be the top item and the store object the second item.



OR



If the top-of-stack item is not a Data Word or an Initial Reference, or if the top item is a Data Word and the second item is not an Initial Reference, an Invalid Stack Argument interrupt is generated. If any reference evaluation produces a tag 3 item or an Indexed Word DD is marked read only, a Memory Protect interrupt is generated. If reference evaluation produces an item which is not otherwise a valid result according to the above chain evaluation rules, an Invalid Reference Chain interrupt is generated.

The store object is written into the target location. Normal store evaluation operators will not write into a location containing an odd tagged word.

Type conversion between double-precision operands and single-word items (single-precision operands, tag 4 words, uninitialized operands) depends on the type of the store object (the store type) and the type associated with the target location (the target type). The target type is determined as follows: if one or more Indexed Word DDs are evaluated, the target type is the element-size value of the last Indexed Word DD; otherwise, the target type is single-word if a single-word item is currently stored in the target location and double-precision if a double-precision operand is in the target location.

If the store type is double-precision and the target type is single word, a SNGL (set to single-precision, rounded) operation is performed on the store object, and the resultant single-precision operand is stored into the target location.

If the store type and the target type are double-precision, both words of the store object are stored into the target locations. If the store type is single-word and the target type is double-precision, the single-word store object is extended to double-precision by changing its tag and appending a second word initialized to 0. Both words of the pair are stored into the target locations.

Where two double-precision words are written, if the second (adjoining) target location contains an odd tagged word, a Memory Protect interrupt is generated.

STOD (Store Delete) P(B8)

A normal store operation is performed. Both the Initial Reference and the store object are deleted from the stack.

STON (Store Non-Delete) P(B9)

A normal store operation is performed. The store object is left on top of the stack, and the Initial Reference is deleted. If the store object has been converted during the normal store operation, it is left on the stack in the converted form.

OVERWRITE OPERATORS

Overwrite operators perform a single evaluation of the Initial Reference and store some item from the stack (the store object) into the resultant target location. There are no restrictions on either the store object or the initial contents of the target location:

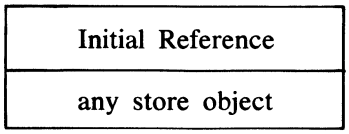
<Initial reference>
    ::= (IRW, Indexed Word DD)

<target>
    ::= (any item)

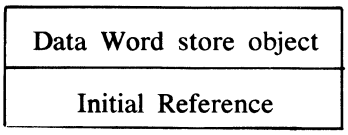
IRW
    --> <target>

Indexed Word DD
    --> <target>

The Initial Reference and the store object are required on top of the stack. The Initial Reference is assumed to be the top item and the store object the second item, but if the top item is a Data Word (even tag), the second item is assumed to be the Initial Reference. A Data Word store object and the Initial Reference may be in either order, but if the store object has an odd tag, the Initial Reference must be the top item and the store object the second item.



OR



If the top-of-stack item is not a Data Word or an Initial Reference, or if the top item is a Data Word and the second item is not an Initial Reference, an Invalid Stack Argument interrupt is generated. If the Initial Reference is an Indexed Word DD marked read only, a Memory Protect interrupt is generated.

Overwrite operators are oblivious to double-precision. If the store object is double-precision, only the first word is stored, with a tag of 2. If an IRW addresses a double-precision item or an Indexed Word DD specifies double-precision, only the first word of the target location is overwritten. Each case may produce an unpaired double-precision word in memory.

OVRD (Overwrite Delete) P(BA)

An overwrite operation is performed. Both the Initial Reference and the store object are deleted from the stack.

OVRN (Overwrite Non-Delete) P(BB)

An overwrite operation is performed. The store object is left unchanged on top of the stack, and the Initial Reference is deleted.

RDLK (Read Lock) V(BA)

RDLK is identical to OVRD except for the stack state at termination. The flashback, the word which was in the memory location into which the overwrite occurred, is left on top of the stack. The flashback is a single word and may be an isolated word of a double-precision item.

SPECIAL EVALUATION OPERATOR

STBR (Step and Branch) P(A4)

If the former STBR operator encoding is encountered, an Undefined Operator interrupt is generated.

PROCESSOR STATE OPERATORS

This section deals with operators which interact with logical processor state, primarily the state of the currently executing code stream and the state of the stack in which the processor is running.

Preliminary Information

CODE STREAM POINTER DISTRIBUTION

The processor code stream pointer is initialized by the distribution of PCW or RCW code stream pointer components according to the following steps:

1. SDLL and SDI are set from the sdll and sdi fields of the PCW or RCW, respectively, and the referenced code segment Descriptor (SD) is fetched. The address of the code segment Descriptor is relative to the updated display register values. If the tag of the code segment Descriptor is not 3, a Code Segment Error interrupt is generated.
2. For PCWs, the PWI and PSI values are verified as follows. If PCW PWI is not in the range (0 to SD segment length–1), an Invalid Index interrupt is generated, and if PCW PSI is not in the range (0 to 5), an Invalid Argument Value interrupt is generated; otherwise, PWI and PSI are set from PCW PWI and PCW PSI, respectively. For RCWs, PWI and PSI are unconditionally set from RCW PWI and RCW PSI, respectively.
3. If the code segment Descriptor is present, PBR is set from SD.address, PLR is set from PBR + SD.segment length–1, and the processor is conditioned to perform the next execution from the new code segment. If the Descriptor is absent, a Presence Bit interrupt is generated. The code stream pointer distribution is still completed in this case. The RCW constructed for the interrupt contains the pointer just distributed, and exit from the interrupt will complete the intended distribution from the then-present segment Descriptor.

The B 5900 confirms that PCW PWI and PCW PSI are in a valid range. RCW PWI and RCW PSI are not checked. Although the RCW is not fully secure, it is initially created by ENTR (enter) from valid PSI and PWI values.

System Stack Control

The reserved stack location (0,2) is assumed to always contain the Stack Vector Descriptor (SVD), an unindexed Data Descriptor for the stack vector array, which contains Descriptors for all stacks on the system. Stack Descriptors are also unindexed Data Descriptors.

A stack is accessed by its stack number as follows: The Stack Vector Descriptor is accessed and indexed by the stack number; If the stack number is not in the range (0 to SVD length–1), an Invalid Index interrupt is generated; otherwise, the stack Descriptor is fetched.

Both Lexical Links and SIRWs reference the base of an activation record by the pair (stack number, displacement). The stack is accessed, and the computation stack Descriptor.address + displacement yields the MSCW address.

BRANCHING OPERATORS

Branching operators provide for altering the code stream pointer component of the processor. These operators may change the point of execution in the current code segment or establish a new code segment with an initial point of execution.

Branches may be conditional or unconditional. Conditional branches alter the code stream pointer or continue sequential execution depending upon a Boolean interpretation of an item in the stack. The item may be of any type; the Boolean interpretation of the item is true if its low order bit is set to 1 and false otherwise. For double-precision items, the low-order bit of the first word is tested.

Branching operators are classified as static or dynamic branches as discussed in the following two sections.

Static Branches

Static branches always branch to a point within the current code segment. That point is indicated by a 2-syllable parameter. Op name stands for the particular static branch operator encoding:

op name	op : psi:	op pwi
	3 :	13

The high-order 3 bits of the parameter are interpreted as the new PSI value, and the low-order 13 as the new PWI value. If the branch is taken and op pwi is not in the range (0 to SD.segment length–1), where SD is the current code segment Descriptor, an Invalid Index interrupt is generated. If the branch is taken and op psi is greater than 5, an Invalid Code Parameter interrupt is generated. If the branch is not taken and either of the above tests fails, the result is undefined.

BRUN (Branch Unconditional) P(A2)

Processor registers PSI and PWI are set from the parameters, and the processor is conditioned to execute next the operator at that point in the current code segment.

BRTR (Branch True) P(A1)

The top-of-stack item is interpreted as a Boolean value. If the value is true, processor registers PSI and PWI are set from the parameters, and the processor is conditioned to execute next the operator at that point in the current code segment. If the value is false, sequential execution continues.

BRFL (Branch False) P(A0)

The top-of-stack item is interpreted as a Boolean value. If it is false, processor registers PSI and PWI are set from the parameters, and the processor is conditioned to execute next the operator at that point in the current code segment. If it is true, sequential execution continues.

Dynamic Branches

Dynamic branches take their code stream pointer values from a branch destination item on top of the stack. They may branch to a computed point within the current code segment or to a point in an arbitrary code segment.

Branching within the current code segment is indicated if the branch destination item is an operand. It is integerized with rounding, if required, to produce a single-precision integer. If the operand cannot be integerized, an Integer Overflow interrupt is generated; otherwise it is interpreted as the code segment index in units of half-words (3 syllables).

dyn pwi [13:13]
The new PWI value

alignment [0:1]
The alignment bit (0=word boundary, 1=half word boundary)

The new PSI value is 0 (the word boundary) if the alignment bit is 0 and 3 (the half-word boundary) if the alignment bit is 1. If the branch is taken and dyn–pwi is not in the range (0 to SD.segment length–1), where SD is the current code segment Descriptor, an Invalid Index interrupt is generated. If the branch is taken and the integer value of the operand is not in the range (0 to 2^{13}), an Invalid Index interrupt is generated. If the branch is not taken and either of the above tests fails, the result is undefined.

Branching to a point in an arbitrary code segment is indicated if the branch destination item is a PCW, or an NIRW chain to a PCW. The PCW code stream pointer is distributed as disclosed as above. PCW control state is ignored.

If the top-of-stack item is not an NIRW, PCW, or operand, an Invalid Stack Argument interrupt is generated. If NIRW chain evaluation does not produce a PCW, an Invalid Reference Chain interrupt is generated (an SIRW is invalid), and if PCW ll is not equal to LL, an Invalid Argument Value interrupt is generated.

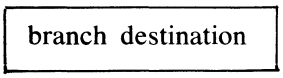
DBUN (Dynamic Branch Unconditional) P(AA)

A branch destination item is required on top of the stack. If it is an operand, the branch is within the current code segment. When the item is a PCW, or an NIRW chain to a PCW, the branch is to an arbitrary code segment.

DBTR (Dynamic Branch True) P(A9)

The second from top-of-stack item is interpreted as a Boolean value. If the value is true, a branch is executed using the top-of-stack branch destination item exactly as in DBUN. If it is false, sequential execution continues.

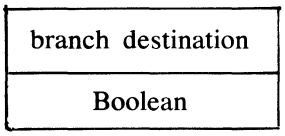
The required initial stack state is:



DBFL (Dynamic Branch False) P(A8)

The second from top-of-stack item is interpreted as a Boolean value. If false, a branch is executed using the top-of-stack branch destination item exactly as in DBUN. If true, sequential execution continues.

The required initial stack state is:



STACK STRUCTURE OPERATORS

Stack structure operators provide for procedure entry and exit and for changing the site of activity of the processor by establishing a new running stack. Stack Structure Operators are involved in setting, saving, and restoring processor state components and linkage of activation records in the stack, both historical and lexical.

PROCEDURE ENTRY OPERATORS

In general, executing a procedure call requires a code sequence which performs the following steps:

1. Execution of MKST (mark stack) initializes the MSCW at the base of the new activation record;
2. The PCW for the procedure, or a reference to the PCW, is pushed onto the stack (in the location which the RCW will subsequently occupy);
3. Parameters to the procedure are built by operators which push items onto the stack (executed in the environment of the caller);
4. Execution of ENTR (enter) completes the stack linkage in the activation record, making it the new topmost activation record, while saving the environment of the caller and instating the environment of the procedure;
5. Stack building code initializes the local variables of the procedure, and the procedure body is executed.

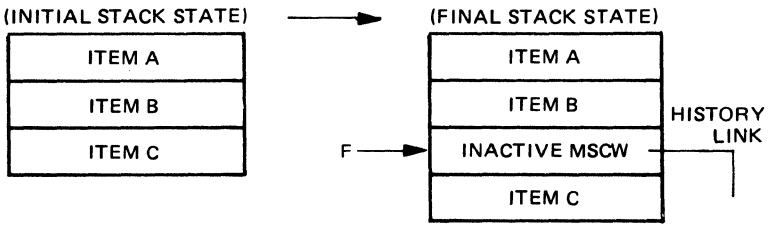
MKST (Mark Stack) P(AE)

MKST builds an inactive MSCW on top of the stack, and inserts it at the head of the historical chain. The History Link field is set to the displacement down the stack to the MSCW addressed by the F register. All other fields are initialized to 0, this includes the entered bit which marks the MSCW inactive. F is set to point to the new MSCW on top of the stack.

IMKS (Insert Mark Stack) P(CF)

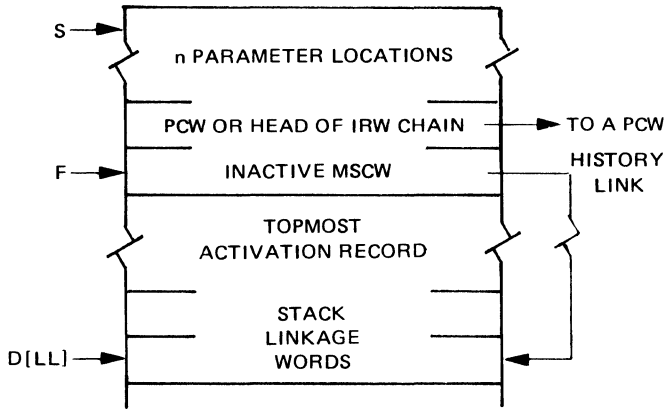
IMKS builds an inactive MSCW exactly as does MKST (mark stack), except that the new MSCW is inserted "underneath" the two top-of-stack items. IMKS produces the effect of having saved the top two stack items, deleted them from the stack, invoked MKST, and then pushed the two items back onto the stack.

The following diagram illustrates the stack state transformation produced by IMKS:



ENTR (Enter) P(AB)

The initial stack state for ENTR assumes prior execution of MKST (or IMKS). An inactive MSCW is required at the stack location addressed by F, and either a PCW or the head of an IRW chain to a PCW is required at the F+1 stack location. The following diagram illustrates the initial stack state:



If the item addressed by F does not have a tag of 3, when the entered bit of the item is 1 (indicating an entered MSCW), or if the top-of-stack address is less than or equal to F, a Stack Structure Error interrupt is generated. If the F+1 stack location is not a PCW or the head of an IRW chain, an Invalid Stack Argument interrupt is generated, and if IRW chain evaluation does not produce a PCW, an Invalid Reference Chain interrupt is generated. If PCW ll is greater than 15, a Stack Structure Error interrupt is generated.

ENTR consists of the following functional tasks:

1. Complete the MSCW and insert it at the head of the lexical chain formerly headed by the immediate global activation record.
2. Construct an RCW which saves the current processor code stream pointer and Boolean accumulators.
3. Initialize processor state for the procedure being entered, including code stream pointer and addressing environment.

(1) Complete the MSCW:

The activation record containing the PCW is the immediate global addressing space (global AR) of the procedure being entered. ENTR completes the lexical chain which defines the procedure's addressing environment by constructing the Lexical Link to point to the global AR. Stack number and displacement are set to address the base word.

The global AR is identified by the form of reference to the PCW (the final reference if an IRW chain is evaluated). If the reference is an NIRW, the global AR is the activation record at level NIRW.Lambda in the addressing environment at invocation of ENTR. If it is an SIRW, stack number and displacement point directly to the global AR. If there is no reference to the PCW (the F+1 stack location initially held the PCW), then the global AR is assumed to be the activation record at level PCW ll-1 in the addressing environment at invocation of ENTR. In this case, if PCW ll is not in the range (1 to LL+1), an Invalid Argument Value interrupt is generated.

PCW ll is copied into MSCW ll, MSCW entered is set to 1, and the completed MSCW is stored back at the F stack location. Note that MSCW restart and MSCW History Link are not altered by ENTR.

(2) Construct the RCW:

Processor state values stored in the RCW are the four Boolean accumulators (TFFF, OFFF, EXTF, and FLTF), the processor code stream pointer (SDLL, SDI, PSI, PWI), Control State (CS), and Lexical Level (LL). The RCW is stored at the F+1 stack location.

(3) Initialize Processor State:

LL is set from PCW.ll and D[LL] is set from F to address the base of the activation record. The processor CS Boolean is set from PCW control state. Any applicable display registers are updated to reflect the addressing environment described by the values of D[LL] and LL.

The processor code stream pointer state is initialized from the PCW.

PROCEDURE EXIT OPERATORS

There are two operators for deleting an activation record and returning execution to the prior topmost activation record. RETN (return) assumes termination of a function and leaves the top-of-stack item as a result, whereas EXIT assumes termination of a procedure and does not leave a result.

EXIT (Exit) P(A3)

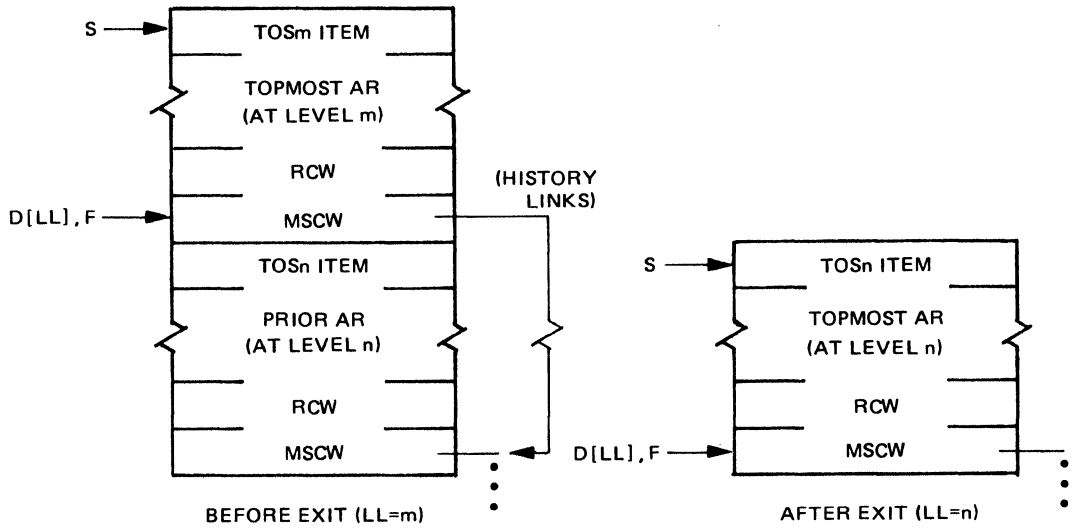
EXIT deletes the topmost activation record from the stack and restores processor state for the prior topmost activation record. The base location of the topmost activation record is addressed by D[LL], and the prior topmost activation record is identified by the first entered MSCW on the historical chain whose head is D[LL].

If D[LL] <= BOSR+1, a Bottom of Stack interrupt is generated, and if the tag of the D[LL] or D[LL]+1 item is not 3, a Stack Structure Error interrupt is generated. Otherwise, the base of the prior topmost activation record is located by following the historical chain from D[LL] until the first entered MSCW is encountered. If a History Link is evaluated which points to a location less than BOSR, or if the tag of a stack item addressed by a History Link is not 3, or if the ll field of the first entered MSCW is not equal to the ll field of the RCW in the initial topmost activation record, a Stack Structure Error interrupt is generated.

The topmost activation record is deleted from the stack by setting the top-of-stack pointer, S, to D[LL]-1. F is reset to address the first MSCW on the historical chain whose head is D[LL], whether or not it is entered. D[LL] is reset to address the base of the prior topmost activation record. Remaining processor state is reset by distributing values saved in the RCW at the initial D[LL]+1 stack location. The four processor Boolean accumulators (TFFF, OFFF, EXTF, FLTF), CS (control state), and ll are reset from their saved values in the RCW. Any applicable display registers are updated to reflect the addressing environment described by the new values of D[LL] and LL.

The processor code stream pointer is initialized from the RCW. If MSCW restart = 1, the processor is conditioned to execute in restart state the operator addressed by the new code stream pointer.

The following diagram illustrates the stack state transformation produced by EXIT. In the initial state, F is shown pointing to the same location addressed by D[LL], but that is not required:



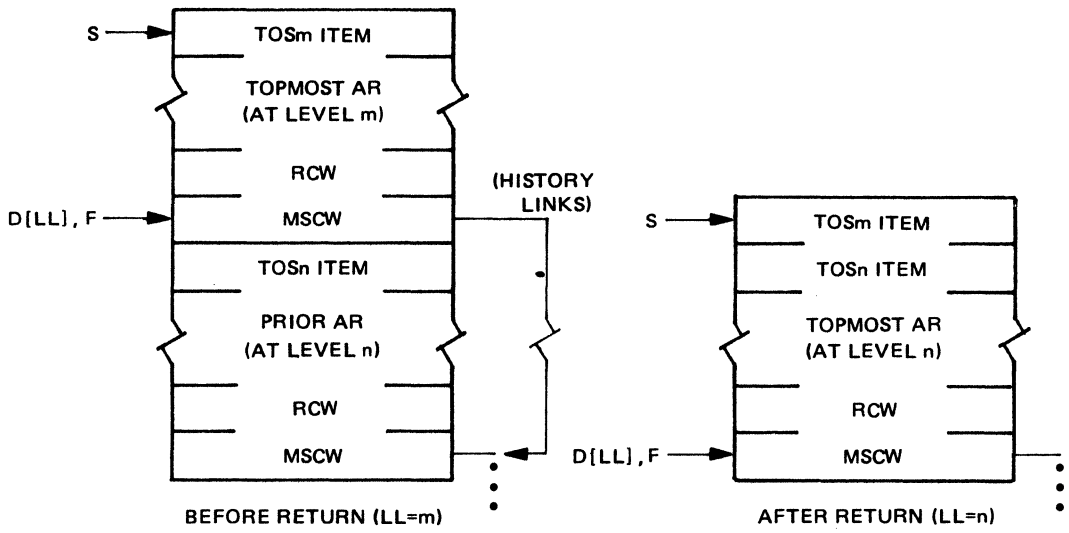
The B 5900 generalizes the MSCW restart mechanism for both EXIT and RETN. If MSCW restart = 1, the processor is conditioned to execute the operator being returned to in a unique restart state.

The B 5900 EXIT requires that the D[LL]+1 stack item have a tag of 3, and that the first entered MSCW on the historical chain be at RCW.ll.

RETN (Return) P(A7)

RETN is exactly the same as EXIT, except that it assumes that the terminated activation record is a function and that the initial top-of-stack item is to be the result of the function. RETN therefore retains the initial top-of-stack item and pushes this item back onto the top of the stack after the topmost activation record is deleted.

The following diagram illustrates the stack state transformation produced by RETN:



The B 5900 implementation of VALC will allow re-entry to a restart state, such that RETN is not required to complete the VALC operation on the top-of-stack item it leaves as the result.

MVST (Move to Stack) V(AF)

MVST changes the site of activity of the processor by deactivating the current stack and activating a destination stack. A Top of Stack Control Word (TSCW) stored at the base location of an inactive stack is used to preserve processor state sufficient to activate the stack.

MVST consists of the following functional tasks:

(1) Deactivate the current stack:

The current stack TSCW is constructed from the four Boolean accumulators (TFFF, OFFF, EXTF, FLTF), Control State (CS), Lexical Level (LL), and relative specifications for S and F register values. S is indicated by stack height, computed from S – BOSR, and F is indicated by S-F displacement. At conclusion of MVST, the current stack is marked inactive by storing the TSCW at its base location.

(2) Identify the destination stack:

MVST requires a single-precision operand on top of the stack to specify the stack number of the destination stack; otherwise, an Invalid Stack Argument interrupt is generated. The operand is integerized with rounding if required. If the operand cannot be integerized, an Integer Overflow interrupt is generated, and if the result is a valid integer but not in the range (0 to Stack Vector Descriptor.length–1), an Invalid Index interrupt is generated. Otherwise, MVST fetches the stack Descriptor and insures that the stack is present.

Collision with another processor, which could be running in the destination stack, is prevented by a "Read with Lock" operation. The processor ID is written into the base location of the destination stack as a single-precision integer, while the read flashback reflects the prior status of the stack. If the flashback item has a tag of 3, it is interpreted as the destination TSCW; otherwise, (when the stack is active), a Stack Structure Error interrupt is generated.

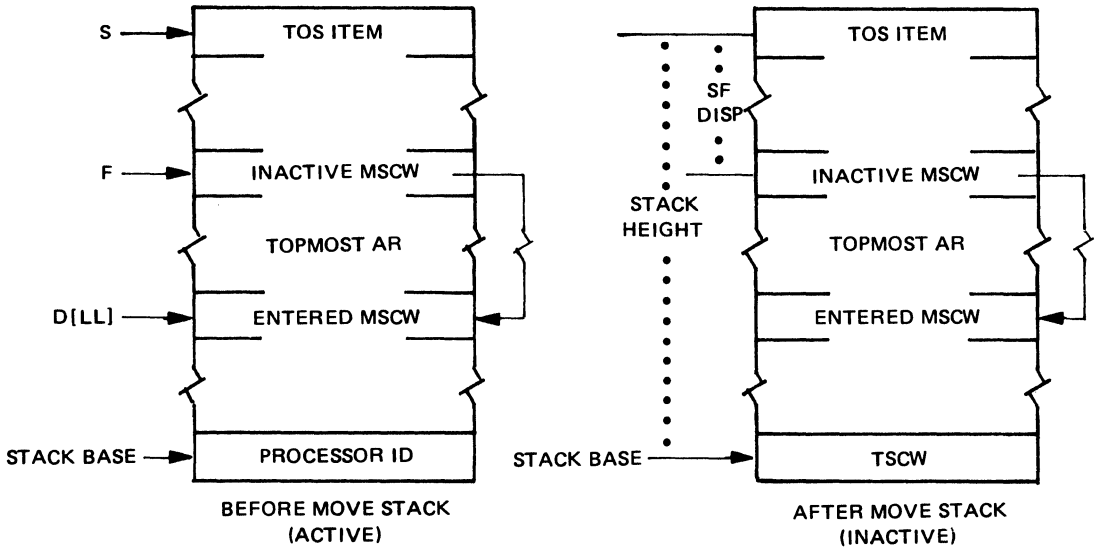
(3) Restore destination stack state:

SNR is set from the stack number, and BOSR and LOSR are set from the stack Descriptor fields address and address+length–1. The TSCW is distributed as follows: 1) The four processor Boolean accumulators (TFFF, OFFF, EXTF, FLTF), Control State (CS), and Lexical Level (LL), are set from their saved values in the TSCW; 2) S is set from TSCW stack height + BOSR and F from S – TSCW SF disp; 3) the resultant F value is less than BOSR, a Stack Structure Error interrupt is generated.

D[LL] is set to point to the first entered MSCW (MSCW entered=1) on the historical chain whose head is F. If in following the historical chain, a History Link is encountered which points to a location less than BOSR, when the tag of a stack item addressed by a History Link is not 3, or if the ll field of the first entered MSCW is not equal to TSCW ll, a Stack Structure Error interrupt is generated.

The display registers are updated to reflect the addressing environment described by the new values of D[LL] and LL.

The following example illustrates the current stack transformation produced by MVST after the destination stack number has been deleted from the stack (the transformation of the destination stack is the opposite of that of the current stack):



NOTE

The code stream pointer state of the processor is absent noticeably from the functioning of MVST. At termination, the code stream bound to the original stack will continue execution. Because of this fact, MVST is executed only within an MCP intrinsic, which consists of the operators MVST and EXIT. The call to the intrinsic preserves the code stream pointer for the current stack in the RCW constructed by ENTR. The EXIT following MVST then establishes the processor code stream pointer for the destination stack by distributing its corresponding RCW.

MVST requires a single-precision operand, which is integerized if required and must produce a valid stack number.

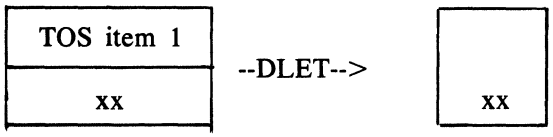
MVST requires that the first entered MSCW on the historical chain from F (in the destination stack) be at the lexical level specified in the TSCW; otherwise a Stack Structure Error interrupt is generated.

TOP-OF-STACK OPERATORS

These operators alter the top-of-stack state, while leaving the remaining processor state unchanged. There is no restriction on the type of stack item which will be acted upon, but as operator arguments, the items must be at or above D[LL]+1.

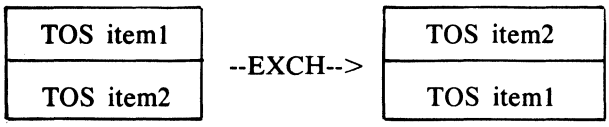
DLET (Delete Top-Of-Stack) P(B4)

DLET requires one item on top of the stack and deletes it from the stack:



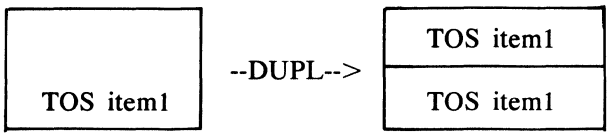
EXCH (Exchange Top-Of-Stack) P(B6)

EXCH requires two items on top of the stack and interchanges their order in the stack:



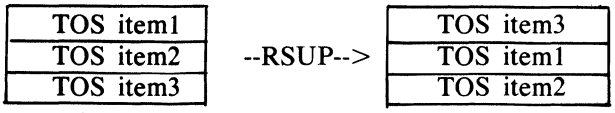
DUPL (Duplicate Top-Of-Stack) P(B7)

DUPL requires one item on top of the stack, creates an exact copy, and leaves both items on top of the stack:



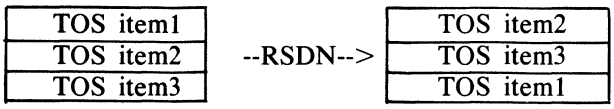
RSUP (Rotate Stack Up) V(B6)

RSUP requires three top-of-stack items. The third from top item is "rotated up" to become the top of stack item:



RSDN (Rotate Stack Down) V(B7)

RSDN requires three top-of-stack items. The top item is "rotated down" to become the third from top of stack item:



PROCESSOR STATE MANIPULATION OPERATORS

This set of operators either places processor state register values on top of the stack, sets registers from top-of-stack item values or performs a combination of the two. Operators are classified as read state, set state, or read and set state.

READ STATE OPERATORS

RTFF (Read True-False Flip-Flop) P(DE)

RTFF leaves on top of the stack a single-precision word with the value of TFFF (True-false Flip-flop) in the low-order bit. The high-order 47 bits of the result are 0.

RCMP (Read Compare Flip-Flop) V(B3)

If the former RCMP operator encoding is encountered, an Undefined Operator interrupt is generated.

WHOI (Read Processor ID) V(4E)

WHOI leaves on top of the stack a single-precision word containing the processor identification number.

RTOD (Read Time of Day Clock) V(A7)

RTOD leaves on top of the stack a single-precision integer containing the value of the time of day clock. The range of values is (0 to 2^{35}) in units of 2.4 microseconds.

RPRR (Read Processor Register) V(B8)

RPRR requires a single-precision operand on top of the stack; otherwise, an Invalid Stack Argument interrupt is generated. The operand is interpreted as a processor register identification (register ID), and the result left on top of the stack is a single-precision integer whose value is that of the specified register.

Readable processor registers are associated with register IDs which are a subset of integers in the range (0 to 63). The register ID operand is integerized with rounding, if required. If the operand cannot be integerized, an Integer Overflow interrupt is generated, and if the result is a valid integer but not a valid register ID value, an Invalid Argument Value interrupt is generated. Table 4-1 specifies the decimal register ID encoding.

Table 4-1. Decimal Register ID Encoding

Decimal Register ID	Register Name	Status
0, LL	D[0], D[LL]	OK
32	PWI	OK
36	LOSR	OK
37	BOSR	OK
38	F	OK
48	PBR	OK
52	S	OK
53	SNR	OK
54	SDLL&SDI	OK

RPRR requires an operand to specify the register ID, and it must evaluate to a valid register ID value. The result is a single-precision integer having the value of the specified register.

If an invalid register ID value is encountered, an Invalid Argument Value interrupt is generated.

The value of S produced by RPRR is the address of the top stack word in memory. Depending on the state of top-of-stack registers, this value may address the logical top, second or third from top of stack item. RPRR will ensure that the value of S produced will address the logical top-of-stack item by pushing any items in top-of-stack registers (except its parameter) into memory before returning the value of S.

SET STATE OPERATORS

The following paragraphs define the Set State Operators.

SXSN (Set External Sign Flip-Flop) P(D6)

SXSN sets EXTF (external sign flip-flop) to the value of bit 46 of the top-of-stack item. There is no restriction on the type of the top item, and it is left on the stack unchanged. (Bit 46 is interpreted as the mantissa sign of an arithmetic operand).

EEXI (Enable External Interrupts) V(46)

EEXI conditions the processor to respond to external interrupts and resets the processor CS Boolean to 0 (normal state).

DEXI (Disable External Interrupts) V(47)

DEXI conditions the processor to ignore all external interrupts and sets the processor CS Boolean to 1 (control state).

SINT (Set Interval Timer) V(45)

SINT arms and sets the interval timer using the integer value of an operand on the top of the stack. If the item on the top of the stack is not a single-precision operand, an Invalid Stack Argument interrupt is generated.

The operand is integerized with rounding, if required. If it cannot be integerized, an Integer Overflow interrupt is generated. When the result is a valid integer, but not in the range (0 to 2^{10}), an Invalid Argument Value interrupt is generated. Otherwise, the interval timer is set to the resultant value.

WTOD (Write Time of Day Clock) V(49)

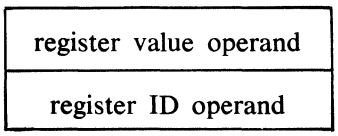
WTOD sets the time of day clock from the integer value of an operand on top of the stack. If the item on top of the stack is not a single-precision operand, an Invalid Stack Argument interrupt is generated.

The operand is integerized with rounding if required. If it cannot be integerized, an Integer Overflow interrupt is generated, and if the result is a valid integer but not in the range (0 to 2^{35}), an Invalid Argument Value interrupt is generated. Otherwise, the time of day clock is set to the resultant value. The clock counts time in units of 2.4 microseconds.

SPRR (Set Processor Register) V(B9)

SPRR requires two single-precision operands on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The top operand is interpreted as a processor register identification (register id), and the contents of the specified register are set to the value of the second operand.

The required initial stack state is:



Setable processor registers are associated with register IDs which are a subset of integers in the range (0 to 63). Both operands are integerized with rounding, if required. If either cannot be integerized, an Integer Overflow interrupt is generated. If the results are valid integers, but the register ID is not a valid value or the register value exceeds the register capacity, an Invalid Argument Value interrupt is generated.

Refer to Table 4-1 for decimal register ID encoding.

If the register ID operand selects D[LL], the register value operand must address a MSCW. If it does not, an Invalid Argument Value interrupt is generated. SPRR requires that both items be operands and that they evaluate to valid values.

SPRR will push items in the top-of-stack registers other than its parameters into memory before altering S. Thus decrementing S will cut back the logical top of stack, and incrementing S will extend the stack, with the new locations at the logical top of the stack.

Setting D[LL] will result in a correctly updated environment. This requires that the new value address an MSCW.

IDLE (Idle Until Interrupt) V(44)

IDLE loops internally until an external interrupt signal is present. At that time, IDLE invokes the interrupt procedure and terminates. The CS flip-flop is not examined or altered.

RUNI (Turn On Running Light) V(41)

RUNI sets the running light indicator, which is reset by the hardware after a non-critical interval.

READ AND SET STATE OPERATOR

Below is the read and set state operator description.

ROFF (Read and Reset Overflow Flip-Flop) P(D7)

ROFF leaves on the top of the stack a single-precision word with the value of OFFF (overflow flip-flop) in its low-order bit. The high-order 47 bits of the result are 0. OFFF is unconditionally reset to 0.

DATA ARRAY OPERATORS

Operators in this group perform functions on arrays specified by Data Descriptor stack arguments. The functions applied generally consist of sequential processing of one or more arrays of word or character elements. Termination occurs when an element length has been exhausted or when a specific condition is satisfied.

Data in one or more of the argument arrays may be modified, the arrays may be processed so that a result is produced, or both actions may occur. Results are indicated by items left on the top of the stack or by the setting of one or more processor Boolean accumulators.

SEARCHING OPERATORS

There are two searching operators. LLLU (linked list lookup) follows an explicitly linked list searching for the first element whose data component is greater than or equal to an argument value, and SRCH (masked search for equal) follows an implicitly ordered list backwards searching for the first word which is bit-wise equal to an argument after both the word and argument have been masked.

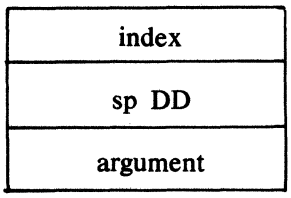
LLLU (Linked List Lookup) V(BD)

LLLU processes an array as an explicitly linked list and applies the following interpretation to each word in the array:

Data Comp [47:28]
: The atomic data component

Link Comp [19:20]
: The link component (an index from the base of the array to the next element in the list)

LLLU requires an initial index, an unindexed single-precision Data Descriptor (DD), and an argument value on top of the stack:



The index and argument must be operands and are integerized by rounding, if required. The element-size field of the DD must be single-precision. If the DD is absent, it will be made present. An Invalid Stack Argument interrupt is generated in the following three cases: if either the index or the argument is not an operand; if the second from top-of-stack item is not a single-precision DD; or if the DD is indexed or paged. If the index or argument cannot be integerized, an Integer Overflow interrupt is generated.

An Invalid Index interrupt is generated whenever an index value is used to fetch a word from the list, if it is not in the range (0 to DD length–1). The initial index is applied to the DD, and the first word of the list is fetched. Starting with that word, LLLU applies the following iterative loop.

If the link component equals 0, a single-precision negative one is left on top of the stack to signal failure. If the link is greater than 0 and the data component is greater than or equal to the absolute value of the argument value, the trail link is left on top of the stack. The trail link is a single-precision integer index of the word whose link points to the current word. If this case occurs for the first word of the list, the index of the first word itself is left.

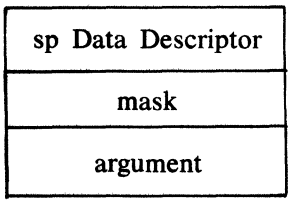
The iteration is repeated for the next word in the list if the link is greater than zero and the data component is less than the absolute value of the argument. Failure termination will occur when the link equals 0 even if the data component of the same word exceeds the absolute value of the argument.

LLLU requires that the argument be an operand and that DD element-size is equal to single-precision.

SRCH (Masked Search for Equal) V(BE)

SRCH scans an array from an indexed starting point back towards the base for a word which is bit-wise equal to an argument value after both have been masked. The masking and comparison are on 51 bits, the word and its tag.

SRCH requires a single-precision Data Descriptor, a mask, and an argument on top of the stack:



The top-of-stack item must be a non paged Data Descriptor whose element-size field specifies single-precision; otherwise an Invalid Stack Argument interrupt is generated. If the Descriptor is an unindexed DD, it is indexed by DD.length–1. If the Descriptor is an unindexed DD and DD length = 0, the search is not done and a single-precision –1 is left of top of the stack to indicate failure.

Both the mask and argument are 51 bit operands. The argument is logically ANDed with the mask before it is used for comparison. The array is scanned from the word addressed by the Indexed DD back to the base of the array. Each word is first logically ANDed with the mask and then compared to the (masked) argument until an equal comparison occurs or the array is exhausted.

If the search succeeds, the single-precision integer index of the matching word is left on top of the stack. If the array is exhausted, a single-precision negative one is left on top of the stack to signal failure.

SRCH requires that the element-size field of the Data Descriptor specify single-precision.

POINTER OPERATORS

Pointer operators provide functions for scanning, transferring, comparing, and editing of a source element sequence or a source and destination element sequence. The sequences consist of word or character elements in data arrays, or of character elements in operands. Sequential processing terminates when an element length is exhausted or in some cases when a condition is satisfied before the length is exhausted.

All pointer operators require initial stack arguments which specify the length and the source element sequence, and most require an argument which specifies the destination element sequence. Each pointer operation includes an "update" operator which leaves on top of the stack at termination a reference to the source, the destination if applicable, and the length if termination is possible before the length is exhausted.

For all transfer operations other than word transfers, the destination tags remain unchanged.

The following sections define requirements and treatment of length, source, destination arguments, and the form of the updated results for each. These definitions are applicable to all pointer operations except word transfer and unpack operations. Exceptions are defined under those operator groups.

Length

The length argument must be an operand; otherwise an Invalid Stack Argument interrupt is generated. It is integerized with rounding when required. If it cannot be integerized, an Integer Overflow interrupt is generated. All length values less than one are equated to zero, and all pointer operators terminate immediately if the length is zero.

An updated length result is a single-precision integer indicating the number of elements remaining to be processed at termination. It is produced by update operators which may terminate before the length is exhausted. If the initial length is negative, the updated length is zero.

Source

The source argument must be an operand, or a DD or Indexed DD of any valid element-size; otherwise, an Invalid Stack Argument interrupt is generated.

A source operand is interpreted according to element-size conventions defined below as a hex or EBCDIC sequence containing 6, 12, or 24 characters (for sp EBCDIC, sp hex, dp EBCDIC, or dp hex respectively). The operand is logically concatenated with itself as required to form an indefinite length sequence. An updated source operand is the original operand circularly rotated left such that the left-justified element is the next element to be processed in the event that termination does not occur.

A source Descriptor is used in the form of a Pointer. If the source argument is a DD, it is indexed by zero to create an Indexed DD no check for invalid index is made. If the element-size of an Indexed DD is single-precision or double-precision (word), it is changed to hex or EBCDIC according to element-size conventions. An updated source Pointer references the next character which would have been processed if termination had

not occured. If the source element-size value is changed to hex or EBCDIC, the updated Pointer contains the modified element-size. If the updated word index value is greater than 2^{15}, an Invalid Index interrupt is generated.

For all pointer operations (except word transfer overwrite), a Paged Array interrupt is generated if an odd-tagged word is read from the source array.

Destination

A destination argument must be a DD or an Indexed DD of any valid element-size; otherwise, an Invalid Stack Argument interrupt is generated. If a destination argument is marked Read-only, operators which store into the destination sequence will generate a Memory Protect interrupt.

A destination Descriptor is used in the form of a Pointer. If the destination argument is a DD, it is indexed by zero to create an Indexed DD. If the element-size of an Indexed DD is word, it is changed to hex or EBCDIC according to element-size conventions. An updated destination Pointer references the next character to be processed in the event that termination does not occur. If the destination element-size value is changed to hex or EBCDIC, the updated Pointer contains the modified element-size. If the updated word index value is greater than 2^{15}, an Invalid Index interrupt is generated.

For all pointer operations which require a destination (except word transfer overwrite), a Paged Array interrupt is generated if a write is attempted into a destination array location which contains an odd-tagged word.

Pointer operators which do not require a destination sequence interpret a source operand as an EBCDIC sequence and change a source word Descriptor to an EBCDIC Pointer. Pointer operators which require both source and destination sequences use the following conventions.

A source operand is interpreted as a hex sequence if the destination argument is a hex Descriptor and as an EBCDIC sequence otherwise. A source word Descriptor is changed to a hex Pointer if the destination argument is a hex Descriptor and to an EBCDIC Pointer otherwise. A destination word Descriptor is changed to a hex Pointer if the source argument is a hex Descriptor and to an EBCDIC Pointer otherwise.

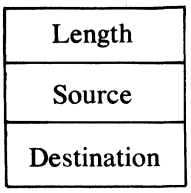
If the source and destination arguments are Descriptors whose element-size values are hex or EBCDIC but not equal, an Invalid Stack Argument interrupt is generated. As with the Word Transfer operators, the TRNS (Translate) operator is an exception to element-size conventions.

The pointer operators treat all source or destination DDs correctly by indexing by zero and do not generate an error interrupt for a paged DD.

CHARACTER TRANSFER OPERATORS

Character transfer operators transfer hex or EBCDIC characters from the source to the destination. The number of characters transferred is specified by the length. TFFF is unconditionally set to 1.

The required initial stack state is:



TUND (Transfer Characters Unconditional Delete) P(E6)

TUND leaves no results on the stack.

TUNU (Transfer Characters Unconditional Update) P(EE)

TUNU leaves the updated source on top of the the stack and the updated destination second from top of the stack.

CHARACTER RELATIONAL OPERATORS

Character relational operators sequentially apply a relational comparison of each source character to a delimiter character supplied by a stack argument until a relation fails or the length is exhausted. TFFF indicates the cause of termination. It is reset to 0 if a relation fails and set to 1 if the length is exhausted (that is, all source characters satisfy the relation).

The delimiter argument must be a single-precision operand; otherwise an Invalid Stack Argument interrupt is generated. It is interpreted as a single right-justified character (EBCDIC if the source is an operand or EBCDIC Descriptor, and hex if the source is a hex Descriptor).

The binary value of the source character is compared to the binary value of the delimiter character. All pointer operators accept any source operand.

SCAN OPERATORS

Character relational scan operators apply the sequential comparison of each source character to the delimiter character as defined earlier in this section.

The required initial stack state is: Delimiter Length Source

The following operators leave no results on the stack:

SGTD (Scan While Greater Delete) V(F2)
SGED (Scan While Greater Than or Equal Delete) V(F1)
SEQD (Scan While Equal Delete) V(F4)
SNED (Scan While Not Equal Delete) V(F5)
SLED (Scan While Less Than or Equal Delete) V(F3)
SLSD (Scan While Less Than Delete) V(F0)

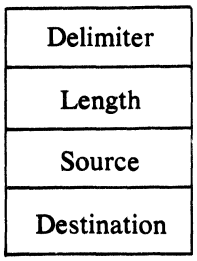
The following operators leave the updated length on top of the stack and the updated source second from top of the stack:

SGTU (Scan While Greater Update) V(FA)
SGEU (Scan While Greater Than or Equal Update) V(F9)
SEQU (Scan While Equal Update) V(FC)
SNEU (Scan While Not Equal Update) V(FD)
SLEU (Scan While Less Than or Equal Update) V(FB)
SLSU (Scan While Less Than Update) V(F8)

TRANSFER OPERATORS

Character relational transfer operators apply the sequential comparison of each source character to the delimiter character as defined above. Each source character which satisfies the relation is transferred to the destination sequence.

The required initial stack state is:



The following operators leave no results on the stack:

TGTD (Transfer While Greater Delete) P(E2)
TGED (Transfer While Greater Than or Equal Delete) P(E1)
TEQD (Transfer While Equal Delete) P(E4)
TNED (Transfer While Not Equal Delete) P(E5)
TLED (Transfer While Less Than or Equal Delete) P(E3)
TLSD (Transfer While Less Than Delete) P(E0)

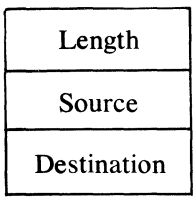
The following operators leave the updated length on top of the stack, the updated source second from top of the stack, and the updated destination third from top of the stack:

TGTU (Transfer While Greater Update) P(EA)
TGEU (Transfer While Greater Than or Equal Update) P(E9)
TEQU (Transfer While Equal Update) P(EC)
TNEU (Transfer While Not Equal Update) P(ED)
TLEU (Transfer While Less Than or Equal Update) P(EB)
TLSU (Transfer While Less Than Update) P(E8)

CHARACTER SEQUENCE COMPARE OPERATORS

Character sequence compare operators apply a relational comparison of the destination sequence to the source sequence. TFFF is set to 1 if the relation of the destination to the source is satisfied and reset to 0 if the relation fails.

The required initial stack state is:



The binary values of each corresponding destination and source character are compared. The destination sequence is equal to the source sequence if each destination character is equal to the corresponding source character or the initial length is zero. The destination is strictly less (greater) than the source, if for the first (leftmost) pair of unequal characters, the destination character is strictly less (greater) than the source character.

The following operators terminate when the actual relation is determined. No results are left on the stack.

CGTD (Compare Characters Greater Delete) P(F2)
CGED (Compare Characters Greater Than or Equal Delete) P(F1)
CEQD (Compare Characters Equal Delete) P(F4)
CNED (Compare Characters Not Equal Delete) P(F5)
CLED (Compare Characters Less Than or Equal Delete) P(F3)
CLSD (Compare Characters Less Than Delete) P(F0)

The following operators terminate only when the length is exhausted. They leave the updated source on top of the stack and the updated destination second from top of the stack (the update forms reference the first character after the end of the sequence as determined by the length).

CGTU (Compare Characters Greater Update) P(FA)
CGEU (Compare Characters Greater Than or Equal Update) P(F9)
CEQU (Compare Characters Equal Update) P(FC)
CNEU (Compare Characters Not Equal Update) P(FD)
CLEU (Compare Characters Less Than or Equal Update) P(FB)
CLSU (Compare Characters Less Than Update) P(F8)

CHARACTER SET MEMBERSHIP OPERATORS

Character set membership operators test source characters for membership in a character set supplied by a stack argument. The relations applied are inclusion and exclusion, and source characters are sequentially tested until the relation fails or the length is exhausted. TFFF indicates the cause of termination: it is reset to 0 if a relation fails and set to 1 if the length is exhausted (that is, all source characters satisfy the relation).

The character set argument must be a Word DD or an Indexed Word DD; otherwise, an Invalid Stack Argument interrupt is generated. If the set argument is a Word DD it is indexed by zero.

The character set is interpreted as a bit array indexed by the source character. If the selected bit is 1, the character is included in the set; otherwise, it is excluded from the set.

The bit array is addressed by adding the indexed address from the character set DD to the WORD INDEX field of the source character that is being checked. The BIT INDEX field of the source character is then adjusted to reference the bit which determines the required relation.

For example: Mem[base address+index + WordIndex(c)].[BitIndex(c):1]

Word Index [7:3]	4	8	Bit Index [4:5]
	2	4	
	1	2	
	16	1	

Bit Format of EBCDIC Character (C)

Word Index = 0	0	8	Bit Index [3:4]
	0	4	
	0	2	
	0	1	

Bit Format of Hex Character (C)

Word Index and Bit Index are computed from the binary representation of the source character (C) as follows:

EBCDIC Word Index
value of 3 high-order bits

Bit Index
31 – (value of 5 low-order bits)

HEX Word Index
0

Bit Index
31 – (4 bit value)

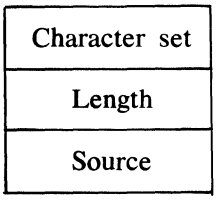
If the set array is paged and the word address computation extends beyond the end of the page, the result is indeterminate.

In the following operator names, the relation "while source included in set" is referred to as "while true", and "while excluded from set" is referred to as "while false".

SCAN OPERATORS

Character set membership scan operators apply the sequential membership test of each source character to the character set as defined previously.

The required initial stack state is:



The following operators leave no results on the stack:

SWTD (Scan While True Delete) V(D5)
SWFD (Scan While False Delete) V(D4)

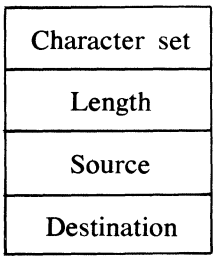
The following operators leave the updated length on top of the stack and the updated source second from top of the stack:

SWTU (Scan While True Update) V(DD)
SWFU (Scan While False Update) V(DC)

TRANSFER OPERATORS

Character set membership transfer operators apply the sequential membership test of each source character to the character set as defined previously Each source character which satisfies the relation is transferred to the destination sequence.

The required initial stack state is:



The following operators leave no results on the stack:

TWTD (Transfer While True Delete) V(D3)
TWFD (Transfer While False Delete) V(D2)

The following operators leave the updated length on top of the stack and the updated source second from the top of the stack, and the updated destination third from the top of the stack:

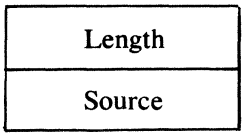
TWTU (Transfer While True Update) V(DB)
TWFU (Transfer While False Update) V(DA)

CHARACTER SEQUENCE EXTRACTION OPERATOR

SISO (String Isolate) P(D5)

SISO extracts a character sequence from the source, creates an operand containing the extracted sequence right-justified with leading 0-fill (if required), and leaves the operand on top of the stack. The length specifies the number of characters in the extracted sequence.

The required initial stack state is:



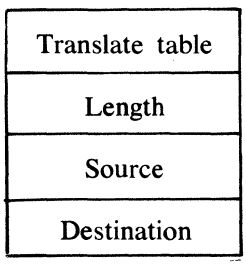
The result may be a single-precision or double-precision operand depending on the length and the source character type. If the source is EBCDIC, the result is single-precision if the length is less than or equal to 6, and and double-precision if length is in the range 7 to 12. If the source is hex, the result is single-precision if the length is less than or equal to 12, and double-precision for 13 to 24. An Invalid Argument Value interrupt is generated if the source is EBCDIC and length is greater than 12, or if the source is hex and length is greater than 24. SISO leaves the resulting operand on top of the stack, except in the case of a Paged Array interrupt, in which case it leaves the updated length on top of the stack, and the updated source second from top of the stack.

CHARACTER SET TRANSLATE OPERATOR

TRNS (Translate) V(D7)

TRNS sequentially accesses characters from the source sequence, maps each character into a specified character set, and stores the translated character into the destination sequence. The character set mapping is indicated by a translate table argument.

The required initial stack state is:



The translate table must be a Word DD or an Indexed Word DD; otherwise, an Invalid Stack Argument interrupt is generated. If it is a Word DD, it is indexed by 0, and no check for invalid index is made. Element-size conventions differ from the other pointer operators. A source operand is interpreted as an EBCDIC sequence, and a source or destination word Descriptor is changed to an EBCDIC Pointer. Source and destination element-size values may differ.

The translate table is interpreted as an array of words, each containing four right-justified eight-bit characters. The translate table is indexed by the source character, and the selected eight-bit character is stored into an EBCDIC destination, or the four low-order bits of the character are stored into a hex destination. The character is indexed by the address equation:

Mem [table address + table index + Word Index (C)] [Field Index (C):8]

Word Index and Field Index are computed from the binary representation of the source character (C) as follows:

EBCDIC Word Index
value of 6 high-order bits

Field Index
8 to the power (4 minus the value of 2 low-order bits) minus 1

HEX Word Index
value of 2 high-order bits

Field Index
8 to the power (4 minus the value of 2 low-order bits) minus 1

If the translate table is paged and the word address computation extends beyond the end of the page, the result is indeterminate.

TRNS always leaves the updated source on top of the stack and the updated destination second from top of the stack.

DECIMAL CHARACTER SEQUENCE OPERATORS

Decimal character sequence operators interpret hex or EBCDIC sequences as decimal sequences, and provide conversion functions among various decimal representations.

A decimal digit is defined to be a hex (four-bit) character in the range hex "0" to hex "9", and a digit sequence is an unsigned sequence of decimal digits. Interpretation of hex or EBCDIC character sequences as decimal sequences and their mapping into corresponding digit sequences is defined in the following paragraphs.

The numeric field (low-order four bits) of each character in an EBCDIC sequence is interpreted as a decimal digit. The corresponding digit sequence is the sequence of numeric fields with the zone fields (high-order four bits) removed. The sign is determined by the zone field of the right-most character. Hex "D" is negative, and any other value is positive.

A hex decimal sequence is signed if the left-most hex character is in the range (hex "A" to hex "F"). Hex "D" is negative, and any other sign value is positive. The corresponding digit sequence is the hex sequence with the left-most sign removed. The length of the hex decimal sequence excludes the sign; thus length+1 hex characters are processed.

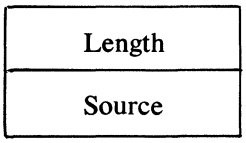
A hex decimal sequence is unsigned if the left-most character is a decimal digit, and the corresponding digit sequence is identical to the hex sequence.

PACK OPERATORS

Pack operators perform a conversion from the source EBCDIC or hex decimal sequence to a decimal operand containing the corresponding digit sequence right-justified with leading 0-fill. The operand is left as a result

on the stack, and TFFF is set to indicate the sign of the source sequence: 1=negative, and 0=positive or unsigned.

The required initial stack state is:



The result is a single-precision operand if length is less than or equal to 12, and double-precision if length is from 13 to 24). If length is greater than 24, an Invalid Argument Value interrupt is generated.

PACD (Pack Delete) P(D1)

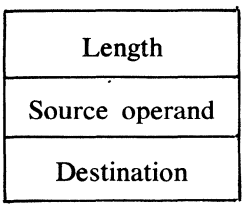
PACD leaves the decimal operand on top of the stack. However, in the case of a Paged Array interrupt, PACD leaves the updated length on top of the stack, the updated source second from the top of the stack, and the partial result third from top of the stack.

PACU (Pack Update) P(D9)

PACU leaves the updated source on top of the stack and the decimal operand second from top of the stack. However, in the case of a Paged Array interrupt, PACU leaves the updated length on top of the stack, the updated source second from top of the stack, and the partial result third from top of the stack.

UNPACK OPERATORS

Unpack operators interpret the source operand as a left-justified digit sequence and store the corresponding hex or EBCDIC decimal sequence into the destination. The required initial stack state is:



If the source is not an operand, an Invalid Stack Argument interrupt is generated. If length is greater than 24, an Invalid Argument Value interrupt is generated.

UNPACK ABSOLUTE OPERATORS

Unpack absolute operators store the destination decimal sequence in an unsigned form. For an EBCDIC destination, the zone field of each character including the right-most is set to hex "F". For a hex destination, the digit sequence is stored with no sign character.

UABD (Unpack Absolute Delete) V(D1)

UABD leaves no results on the stack.

UABU (Unpack Absolute Update) V(D9)

UABU leaves the updated source operand on top of the stack and the updated destination second from top of the stack.

UNPACK SIGNED OPERATORS

Unpack signed operators store the destination decimal sequence in a signed form. The source operand sign is determined by EXTF (External Sign Flip-Flop), where 1=negative and 0=positive.

Hex "D" is used as the negative sign character and hex "C" as the positive sign. For an EBCDIC destination, the sign is inserted into the zone field of the right-most character, and all other zone fields are set to hex "F". For a hex destination, the sign is inserted as the left-most character (length+1 hex characters are transferred).

USND (Unpack Signed Delete) V(D0)

USND leaves no results on the stack.

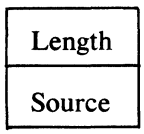
USNU (Unpack Signed Update) V(D8)

USNU leaves the updated source operand on top of the stack and the updated destination second from top of the stack.

INPUT CONVERT OPERATORS

Input convert operators perform a conversion from the source EBCDIC or hex decimal sequence to a numeric operand containing the signed integer value of the corresponding digit sequence. The operand is left as a result on the stack.

The required initial stack state is:



If the length is greater than 23, an Invalid Argument Value interrupt is generated. If the integer absolute value of the source decimal sequence is less than 8^{13}, a single-precision integer is produced, and TFFF is set to 1; otherwise a double-precision integer is produced, and TFFF is reset to 0.

ICVD (Input Convert Delete) P(CA)

ICVD leaves the integer operand on top of the stack. However, in the case of a Paged Array interrupt, ICVD leaves the updated length on top of the stack, the updated source second from top of the stack, and the partial result third from top of the stack.

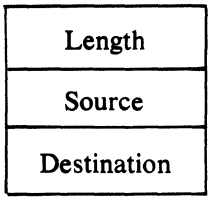
ICVU (Input Convert Update) P(CB)

ICVU leaves the updated source on top of the stack and the integer operand second from top of the stack. However, in the case of a Paged Array interrupt, ICVU leaves the updated length on top of the stack, the updated source second from top of the stack, and the partial result third from top of the stack.

WORD TRANSFER OPERATORS

Word transfer operators transfer word elements from the source to the destination. The number of words is specified by the length. TFFF is unconditionally set to 1.

The required initial stack state is:



A source operand is interpreted as a word or pair of words logically concatenated with itself indefinitely. Source and destination Descriptors are used as Indexed Word DDs. DDs are indexed by 0; Pointers whose Character Index is greater than 0 are changed to point to the next word; that is, word index is incremented by 1 and Character Index is set to 0. An updated source or destination reflects the modified form.

WORD TRANSFER PROTECTED OPERATORS

A word transfer operation is performed as defined above. If an odd-tagged word is read from the source or a store is attempted over an odd-tagged word in the destination, a Paged Array interrupt is generated. Source words are transferred with tags.

TWSD (Transfer Words Delete) P(D3)

TWSD leaves no results on the stack.

TWSU (Transfer Words Update) P(DB)

TWSU leaves the updated source on top of the stack and the updated destination second from top of the stack.

WORD TRANSFER OVERWRITE OPERATORS

Source words are transferred with tags to the destination, regardless of tag value (a Paged Array interrupt cannot occur).

TWOD (Transfer Words Overwrite Delete) P(D4)

TWOD leaves no results on the stack.

TWOU (Transfer Words Overwrite Update) P(DC)

TWOU leaves the updated source on top of the stack and the updated destination second from top of the stack.

EDIT OPERATORS

Edit operators provide for manipulating and editing a source and destination character sequence. The sequences consist of character elements in data arrays or operands. Most edit operators process source and destination characters sequentially until a length is exhausted.

There are two modes in which edit operators are executed, and each is initiated by an Enter Edit operator. The Enter Edit operator provides the source and destination. The operator may specify update, which causes a reference to the source and destination (if applicable) to be left on top of the stack at termination of edit mode.

Table Edit Mode

A sequence of edit operators is executed until terminated by ENDE (end edit). Each acts on the source and destination supplied by the table enter edit operator, and length is a parameter for each edit operator requiring it. If update is specified, the updated source and destination are left on the stack by ENDE.

Single Edit Mode

A single edit operator acts on the source and destination supplied by the enter single edit operator. Length is also supplied at entry, whether or not it is required by the edit operator. If update is specified, the updated source and destination are left on the stack at termination of the edit operator.

The following paragraphs define requirements and treatment of source, destination arguments, and the form of the updated results for each. These definitions are applicable to both Table and Single edit mode.

Source

The source argument must be an operand, or a DD or Indexed DD of any valid element-size; otherwise an Invalid Stack Argument interrupt is generated.

A source operand is interpreted according to element-size conventions defined below as a hex or EBCDIC sequence containing 6, 12, or 24 characters (for single-precision EBCDIC, single-precision hex and double-precision EBCDIC, or double-precision hex respectively). An updated source operand is the original operand circularly rotated left such that the left-justified element is the next element which would have been processed if termination had not occured.

A source Descriptor is used in the form of a Pointer. If the source argument is a DD, it is indexed by zero to create an Indexed DD and no check for invalid index is made. If the element-size of an Indexed DD is single-precision or double-precision (word), the element-size is changed to hex or EBCDIC according to element-size conventions. An updated source Pointer references the next character to be processed if termination does not occured. If the source element-size value is changed to hex or EBCDIC, the updated Pointer contains the modified element-size. If the updated word index value is greater than 2^{15}, an Invalid Index interrupt is generated.

For all edit operators, a Paged Array interrupt is generated if an odd-tagged word is read from the source array.

Each edit operator which uses the source internally updates it at termination, such that a group of edit operators (in table edit mode) sequentially process the source characters. Character skip operators may advance or back up the source to alter the normal sequential processing.

Destination

A destination argument must be a DD or an Indexed DD of any valid element-size; otherwise, an Invalid Stack Argument interrupt is generated. If it is marked read only, edit operators which store into the destination sequence will generate a Memory Protect interrupt.

A destination Descriptor is used in the form of a Pointer. If the destination argument is a DD, it is indexed by zero to create an Indexed DD. If the element-size of an Indexed DD is word, it is changed to hex or EBCDIC according to element-size conventions. An updated destination Pointer references the next character which would have been processed if termination had not occured. If the destination element-size value is changed to hex or EBCDIC, the updated Pointer contains the modified element-size. If the updated word index value is greater than 2^{15}, an Invalid Index interrupt is generated.

For all edit operators, a Paged Array interrupt is generated if a write is attempted into a destination array location which contains an odd-tagged word.

Each edit operator (except skip source characters) internally updates the destination at termination, such that a group of edit operators (in table edit mode) sequentially process destination characters. Character skip operators may advance or back up the destination to alter the normal sequential processing.

A source operand is interpreted as a hex sequence if the destination argument is a hex Descriptor and as an EBCDIC sequence otherwise. A source word Descriptor is changed to a hex Pointer if the destination argument is a hex Descriptor and to an EBCDIC Pointer otherwise. A destination word Descriptor is changed to a hex Pointer if the source argument is a hex Descriptor and to an EBCDIC Pointer otherwise.

If the source and destination arguments are Descriptors whose element-size values are hex or EBCDIC but not equal, an Invalid Stack Argument interrupt is generated.

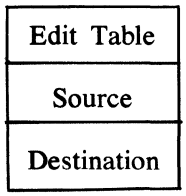
ENTER EDIT MODE OPERATORS

The following paragraphs discuss Enter Edit Mode operators.

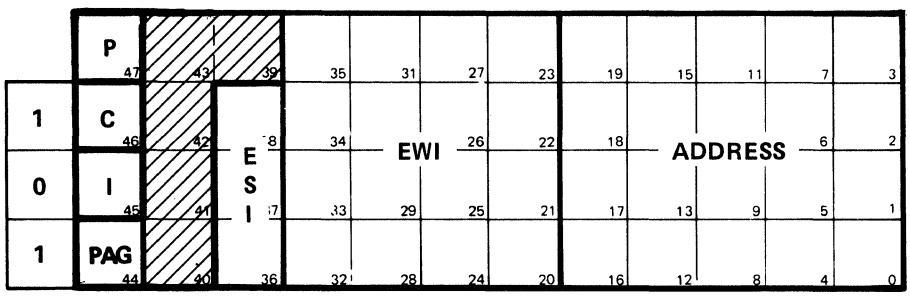
Enter Table Edit Operators

Enter Table Edit operators supply the source and destination sequences, and a reference to the sequence, or table, of edit operators to be executed. Each edit operator acts on either the source, destination, or a combination of the two, supplied at entry, and length is a parameter for each edit operator requiring it.

The required initial stack state is:



If the Edit Table argument is not a Data Descriptor, an Invalid Stack Argument interrupt is generated; otherwise it is interpreted as follows:



P [47: 1]
Presence bit

C [46: 1]
Copy bit

I [45: 1]
Indexed bit

PAG [44: 1]
Paged bit

[43: 5]
Not used

ESI [38: 3]
Edit table syllable index of the first edit operator

EWI [35:16]
Edit table word index of the word containing the first edit operator

ADDRESS [19:20]
Address

If the table Descriptor is unindexed, it is indexed by zero no check for invalid index is made. If the ESI field is not in the range of 0 to 5, an Invalid Argument Value interrupt is generated. If the Descriptor is marked as paged, an Invalid Stack Argument interrupt is generated. Otherwise, Edit operators are fetched from the edit table, starting from the ESI syllable of the EWI word, until completion of an ENDE (End Edit) operator. The normal code stream is then resumed with the operator following the Enter Table edit operator.

If any word fetched from the edit table does not have a tag value of 0 or 3, an Invalid Program Word interrupt is generated. The length of the edit table may not be known and no length check is applied. If an ENDE is not encountered before the table array page is exhausted, results are indeterminate.

TEED (Table Enter Edit Delete) P(D0)

For the TEED operator, the edit mode terminator ENDE leaves no results on the stack.

TEEU (Table Enter Edit Update) P(D8)

For the TEEU operator, the edit mode terminator ENDE leaves the updated source on top of the stack and the updated destination second from top of the stack.

In the case of a Paged Array interrupt, any operator executed in Table Edit mode will leave the updated length on top of the stack, the updated edit pointer second from top of the stack, the updated source pointer third from top of the stack, and the updated destination pointer fourth from top of the stack.

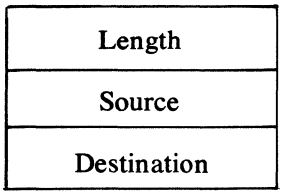
Enter Single Edit Operators

Enter Single Edit operators supply the source and destination sequences and the length for the edit operator which follows it in the code stream. Each argument must be on the stack and must meet type restrictions, although an argument may not be required by the edit operator.

The length argument must be an operand; otherwise, an Invalid Stack Argument interrupt is generated. The operand is integerized with rounding if required. If it cannot be integerized, an Integer Overflow interrupt is generated. All length values less than one are equated to zero, and all edit operators requiring length terminate immediately if it is zero.

Normal Enter Single Edit Operators

These enter single edit operators require length, source, and destination on top of the stack:



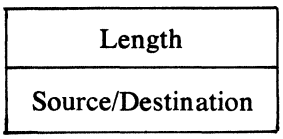
EXSD (Execute Single Edit Operator Delete) P(D2)

For the EXSD operator, the subsequent edit operator leaves no results on the stack.

EXSU/EXPU (Execute Single Edit Operator, Update/Pointer Update) P(DA)/P(DD)

For the EXS and EXPU operators, the subsequent edit operator leaves the updated source on top of the stack and the updated destination second from top of the stack.

EXPU requires length and a single Descriptor specifying both source and destination on top of the stack.



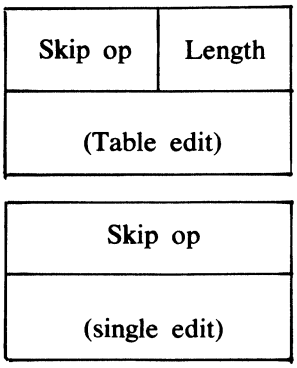
Treatment of the Source/Destination argument and restrictions on its type are the same as those applied to the destination by other Enter Edit mode operators. The element-size convention applied is that a single-precision or double-precision Descriptor is changed to an EBCDIC Pointer.

The subsequent Edit operator leaves the updated Source/Destination Pointer on top of the stack.

CHARACTER SKIP OPERATORS

Character skip operators advance or back up the source or destination sequence. Length indicates the number of characters to be skipped, and the direction is statically indicated by the particular operator. A read only source or destination may be skipped. A negative length argument is equated to zero.

Length is a parameter for Table Edit mode only:



Skip Forward

Character skip forward operators advance the source or destination sequence. A Pointer is incremented by length characters. Each word in the array from the initial to the final point is accessed, and a Paged Array interrupt is generated if a word has an odd-tag. A source operand is circularly rotated left by length characters. If the operator SFSC is entered by the EXPU operator, an Undefined Operator interrupt is generated.

SFSC (Skip Forward Source Characters) E(D2)
SFDC (Skip Forward Destination Characters) E(DA)

Skip Reverse

Character skip reverse operators back up the source or destination sequence. A Pointer is decremented by length characters. If the resultant word index is less than zero, a Paged Array interrupt is generated. If a Paged Array interrupt is generated, the updated version of the pointer which caused the fault has a word index of 2^{15} and a character index of 0. A source operand is circularly rotated right by length characters. If the operator SRSC is entered by the EXPU operator, an undefined Operator interrupt is generated.

SRSC (Skip Reverse Source Characters) E(D3)
SRDC (Skip Reverse Destination Characters) E(DB)

CHARACTER INSERT OPERATORS

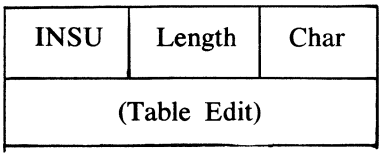
Character insert operators store a character or a sequence of characters into the destination sequence, in some cases conditionally based on the value of FLTF (float flip-flop) and EXTF (external sign flip-flop). Each character is a parameter, except for a fixed sign character.

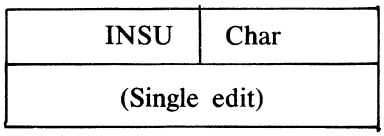
If the destination is marked read only, a Memory Protect interrupt is generated. Several insert operators do not allow a hex destination. Insert operators that allow a hex destination, store only the numeric field of a parameter character.

INSU (Insert Unconditional) E(DC)

INSU stores a sequence composed of length repetitions of the parameter character (Char) into the destination.

Length is a parameter for table edit mode only:

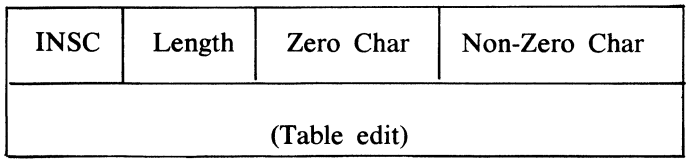


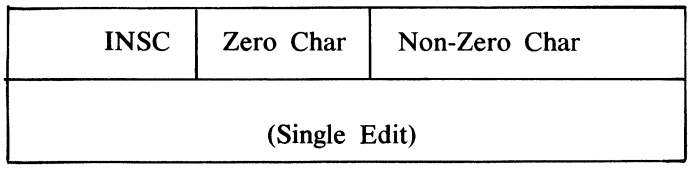


INSC (Insert Conditional) E(DD)

INSC stores a sequence composed of length repetitions of a selected parameter character into the destination. If FLTF=0, Zero Char is selected; if FLTF=1, NonZero Char is selected.

Length is a parameter for Table Edit mode only.





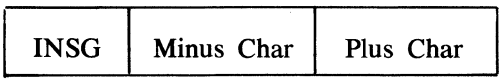
INOP (Insert Overpunch) E(D8)

INOP stores hex "D" into the zone field of the destination character if EXTF=1; the destination character is not altered if EXTF=0. In either case the destination Pointer is advanced 1 character. If the destination element-size is hex, an Invalid Stack Argument interrupt is generated.

INSG (Insert Display Sign) E(D9)

INSG stores Minus Char into the destination if EXTF=1, and stores Plus Char if EXTF=0. If the destination element-size is hex, an invalid Stack Argument interrupt is generated.

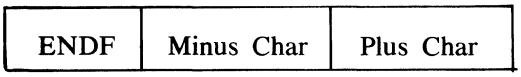
Minus Char and Plus Char are parameters.



ENDF (End Float) E(D5)

If FLTF=0, ENDF stores a selected parameter character into the destination; Minus Char is selected if EXTF=1, and Plus Char is selected if EXTF=0. If FLTF=1, no character is stored, and the destination Pointer is not advanced. FLTF is unconditionally reset to ZERO.

Minus Char and Plus Char are parameters.



CHARACTER MOVE OPERATORS

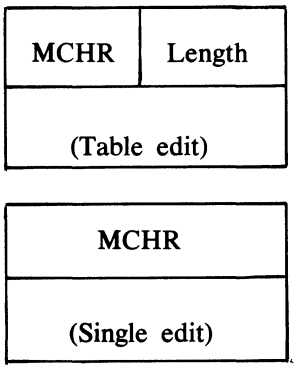
Character move operators transfer characters from source to destination with editing. They may conditionally store into the destination a sequence of repeated parameter characters based on the value of FLTF (Float Flip-flop), the source character, and EXTF (External Sign Flip-flop).

If a Character Move operator is entered by the EXPU operator, an Undefined Operator interrupt is given.

If the destination is marked read only, a Memory Protect interrupt is generated. If the destination element-size is hex, only the numeric field of a parameter character is stored.

MCHR (Move Characters) E(D7)

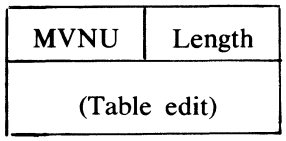
MCHR transfers length characters from the source to the destination. Length is a parameter for Table Edit mode only.

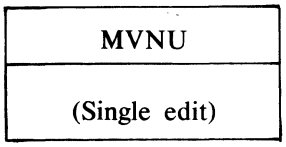


MVNU (Move Numeric) E(D6)

For an EBCDIC source and destination, MVNU transfers length numeric fields from the source to the destination, setting each zone field to hex "F". For a hex source and destination, MVNU transfers length hex characters (in this case MVNU is identical to MCHR).

Length is a parameter for Table Edit mode only.



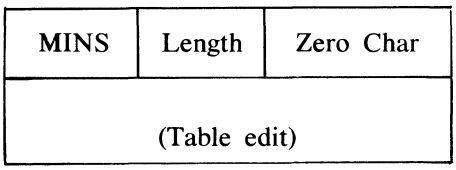


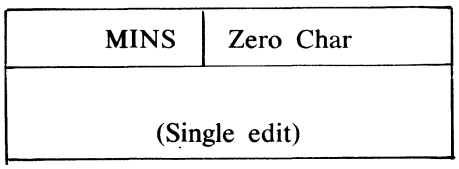
MINS (Move with Insert) E(D0)

MINS performs a leading zero suppression function from the source to the destination for length characters. In the following definition, the "source numeric field" is the numeric field of an EBCDIC character or the entire hex character.

While FLTF=0 and the value of the source numeric field is zero, the Zero Char parameter is transferred to the destination. If the value of the source numeric field is nonzero, FLTF is set to 1, and while FLTF=1, the source numeric field is transferred to the destination. In the latter case, the zone field of each EBCDIC destination character is set to hex "F".

Length is a parameter for Table Edit mode only.





MFLT (Move with Float) E(D1)

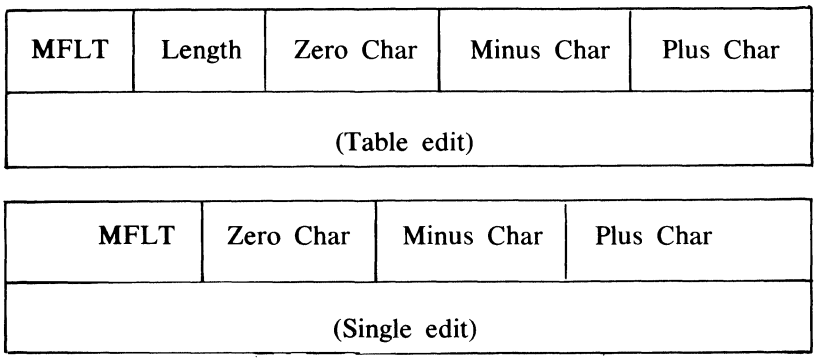
MFLT performs a signed leading zero suppression function from the source to the destination for length source characters. MFLT is functionally equivalent to MINS (Move With Insert) except for conditional insertion of a sign character into the destination sequence.

While FLTF=0 and the value of the source numeric field is zero, the Zero Char parameter is transferred to the destination. If FLTF=0 and the value of the source numeric field is nonzero, FLTF is set to 1, the Plus Char or the Minus Char is inserted in the destination sequence, and the source numeric field is transferred as in MINS. If EXTF=1, the Minus Char is selected and if EXTF=0, the Plus Char is selected.

While FLTF=1, the source numeric field is transferred to the destination, as in MINS.

The number of characters stored into the destination sequence is normally length+1. Length characters are stored only if FLTF is initially 0 and for length characters, all source numeric fields are zero, or if FLTF is initially 1.

Length is a parameter for Table Edit mode only.



MISCELLANEOUS OPERATORS

The following operators are not generally interpreted as belonging to a specific group.

RSTF (Reset Float Flip-Flop) E(D4)

RSTF unconditionally resets FLTF (Float Flip-flop) to 0.

ENDE (End Edit) E(DE)

ENDE terminates Table Edit mode. If update was enabled by the Enter Edit operator, ENDE leaves the updated source on top of the stack and the updated destination second from top of the stack.

Pointers are updated as if a zero length operation occurred. In particular, string DDs will be indexed and word DDs will be changed to pointers.

NOOP (No Operation) P(FE) V(FE) E(FE)

No action is performed.

PUSH (Push Down Stack Registers) P(B4)

The contents of the two top-of-stack registers are pushed onto the memory stack. This action is necessary to make the items in the top of stack registers addressable.

HALT (Conditional Processor Halt) P(DF) V(DF) E(DF)

HALT causes processor execution to halt if the processor Halt Boolean is enabled. If it is disabled, HALT is identical to NOOP (no operation).

NVLD (Invalid Operator) P(FF) V(FF) E(FF)

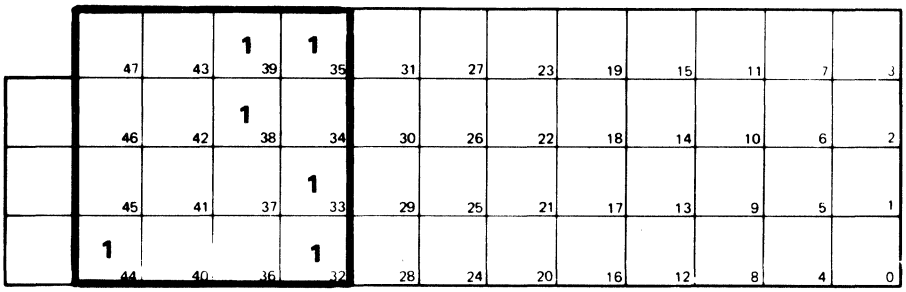
An Invalid Operator interrupt is unconditionally generated.

EXTERNAL COMMUNICATION OPERATORS

CUIO (Communicate with Universal I/O) V(4C)

CUIO requires an Input/Output Control Block (IOCB) data Descriptor on top of the stack and passes the address field of the IOCB Descriptor to the Message Level Interface Port (MLIP). An IOCB Descriptor must be an unindexed, non-paged word Descriptor. The first word of the referenced IOCB array must be a single-precision operand containing an IOCB mark, hex "10CB", in the field [47:16].

If the top-of-stack item is not a valid IOCB Descriptor, an Invalid Stack Argument interrupt is generated. If the first word of the IOCB array is not a single-precision operand with a valid IOCB mark, an Invalid Argument Value interrupt is generated. Otherwise, the IOCB Descriptor address field is transmitted to the MLIP, and CUIO terminates when the MLIP acknowledges receiving the address.



IOCB MARK

SCNI (Scan In) V(4A)

SCNI requires a single-precision operand on top of the stack; otherwise, an Invalid Stack Argument interrupt is generated. The address field ([19:20]) of the operand is passed to the Global System Control (GSC) of the Global Memory Module as a scan in request. The full word response from the GSC is left on top of the stack.

The request sub-field [19:4] must equal binary "1011". If the request bit [15:1] is 0, the GSC response will be from the Response Buffer; if it is 1, the response will be from the Message Buffer. The request is checked by the GSC; if invalid, the result will be a Global Memory Uncorrectable Error for invalid address.

SCNO (Scan Out) V(4B)

SCNO requires two single-precision operands on top of the stack; otherwise, an Invalid Stack Argument interrupt is generated. The address field ([19:20]) of the top-of-stack operand is passed to the Global System Control (GSC) of the Global Memory Module as a scan out request, and the second from top-of-stack operand is passed as the scan out data.

The request sub-field [19:5] must equal binary "10110". The request is checked by the GSC; if invalid, the result will be an Invalid Address interrupt.

INTERRUPTS

PRELIMINARY INFORMATION

The B 5900 invokes and passes information directly to the software operating system (the MCP) by use of interrupts. Depending on the nature of the interrupt, the code stream generating the interrupt will or will not be resumed subsequent to MCP processing of the interrupt.

There are three classes of interrupts:

1. Operator Dependent interrupts: Invoked directly by the current operator to request an MCP service required by the operator or to report a programming or operator fault;
2. Alarm interrupts: Triggered by hardware fault detection during operator execution;
3. External interrupts: Invoked between operators to report events which are independent of the executing code stream.

Alarm and External interrupts are asynchronous with operator execution, indicating respectively hardware fault and MCP attention conditions, whereas ODI conditions are detected directly by operators or from parsing of the code stream.

INTERRUPT ENTRY SEQUENCE

An interrupt is implemented as a parameterized accidental entry to a fixed MCP procedure. That procedure must be visible at the address couple (0,3) in the addressing environment for any executing code stream. Exit from the MCP interrupt procedure will return execution to the interrupted code stream.

Generation of the interrupt procedure entry parallels a normal procedure entry sequence. (See Procedure Entry Operator heading.) An inactive MSCW is built on the stack, with its History Link pointing to the base of the current topmost activation record (F register contents). An NIRW containing the fixed address couple (0,3) and the two interrupt parameter words (see Figure 4-1) are pushed onto the stack above the inactive MSCW. The interrupt procedure entry is completed by invoking the ENTR (enter) operator.

Figure 4-1 illustrates the stack state transformation produced by the interrupt entry sequence (F is shown pointing to the same location as D[LL] in the initial state, but that is not required).

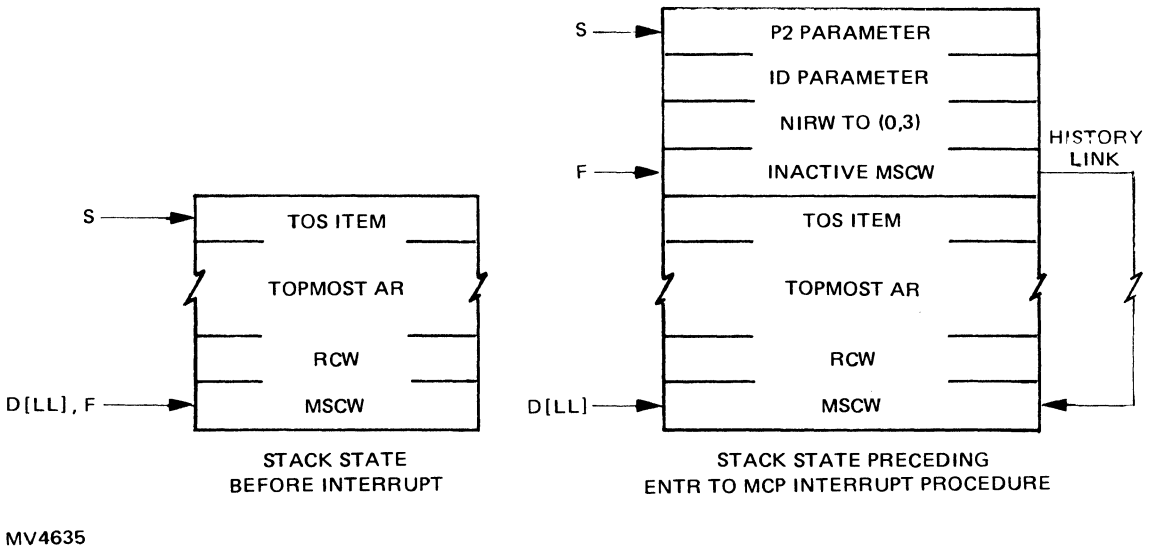


Figure 4-1. Stack State Transformation Produced By Interrupt Entry

For External interrupts, the RCW created by ENTR (and stored at the F+1 stack location) points to the next operator in the current code stream, and for Alarm interrupts, it points to the operator which was executing when the fault was detected. For most Operator Dependent interrupts, the RCW will point to the operator which generated the interrupt. In Single Edit mode, the executing operator is considered to be the Enter Single Edit operator, not the Edit operator; similarly, for variant operators the "95" operator is pointed at. The only exception is for a subset of Presence Bit interrupts (see discussion under MCP Service heading).

Interrupt Parameters

Information passed to the MCP interrupt procedure is contained in two parameter items in the stack. The first is a single-precision operand interpreted as an interrupt identification literal (ID). The second item, called the P2 parameter, varies according to the nature of the interrupt. ID interpretation differs from P1 (it is generally simplified with fewer special state bits), P2 passes a relevant diagnostic item. For Alarm memory errors, the memory address is passed in the P2 parameter, but the direct operator identification and operator state are not reported.

ID PARAMETER

The first interrupt parameter is the single-precision interrupt identification literal (ID parameter). For all interrupts, bit (ID [28:1]) specifies an interrupt, and the interrupt (int) class field (ID [26:3]) indicates the class of interrupt with values (1=Operator Dependent, 2=Alarm, 4=External). The values 0, 3, 5, 6, and 7 are invalid.

Further interpretation of the ID parameter depends on the interrupt class value and is defined under the Interrupt Definition heading with each class of interrupt.

Interpretation of the low-order interrupt type field of the ID parameter depends on the interrupt class. The ODI interrupt type is vertically encoded; each value from 0 to a maximum interrupt type value identifies a specific interrupt.

P2 PARAMETER

The second interrupt parameter, the P2 parameter, varies according to the specific interrupt type. For a given interrupt, the P2 parameter will be an item of fixed or varying type, or the P2 parameter will contain no information.

Interrupt Definition will specify interpretation individually for all interrupts in which P2 is meaningful. Where the P2 parameter contains no information, it will not be explicitly specified.

Superhalt

Special detection exists to prevent a non-terminating interrupt loop. For example, such a loop could be generated and continued indefinitely if the ENTR operator (invoked by the interrupt entry sequence) generates an interrupt which subsequently invokes the ENTR to generate an interrupt.

An effectively non-terminating interrupt loop, called a Superhalt condition, is defined to be a sequence of more than four interrupt invocations without an EXIT or RETN operator being executed during that sequence. When a Superhalt condition is detected, the maintenance console is notified, and the processor is immediately halted.

INTERRUPT DEFINITION

The following paragraphs discuss the three interrupt definitions.

Operator Dependent Interrupts

Operator dependent interrupts are invoked directly by the current operator to request an MCP service required by the operator or to report a programming or operator fault. The RCW created by the interrupt entry will point to the operator generating the interrupt except for a subset of Presence Bit interrupts.

The Operator Dependent ID parameter identifies the type of interrupt, and indicates by the retry bit whether or not the interrupted code stream may be resumed subsequent to MCP interrupt processing. Retry is set if the stack state at the time of the interrupt is still consistent with the required state for operator retry. That state may be the initial or restart state as noted by the restart bit in the interrupt MSCW. ID also includes a state bit, P2 double, which indicates that the P2 parameter is a single-precision first word of a double-precision item.

Operator Dependent ID (int class = 1)

E mode bit [28:1]
Constant value 1

int class [26:3]①
Constant value 1 = Operator Dependent

retry [19:1]
If 1, the stack state is still consistent with the state required for operator retry. If 0, stack arguments have been consumed; an attempt to retry will produce an error condition.

P2 double [18:1]
If 1, the P2 parameter is a single-precision first word of a double-precision item. If 0, the tag of the P2 parameter correctly indicates its type.

int type [4:5]
The type of Operator Dependent interrupt, where:

0 = Presence Bit
1 = Paged Array
2 = Stack Overflow
3 = Invalid Operator
4 = Undefined Operator
5 = Invalid Stack Argument
6 = Invalid Argument Value
7 = Invalid Code Parameter
8 = Invalid Reference
9 = Invalid Reference Chain
10 = Invalid Index
11 = Memory Protect
12 = Divide by Zero
13 = Exponent Underflow
14 = Exponent Overflow
15 = Integer Overflow
16 = Stack Underflow
17 = Bottom of Stack
18 = Stack Structure Error
19 = Code Segment Error
20 = Invalid Program Word

Operator Dependent interrupts fall into two classes: those which request an MCP service, and those which report error conditions arising from programming or operator faults.

MCP SERVICE

Presence Bit, Paged Array, and Stack Overflow interrupts are requests for an MCP service which is an extension of the hardware operators. For all such interrupts, ID retry=1, and in the normal case, the operator generating the interrupt will be resumed subsequent to MCP interrupt processing.

Presence Bit

Presence Bit is used by operators to gain access to a data array or program code segment which is not present in memory. An element of a data array is accessed through an Indexed DD, which may be directly encountered by an operator or may be the result of a dynamic indexing operation on an Un-indexed DD. A program code segment is accessed through a segment Descriptor.

A Presence Bit interrupt is generated when access is required under the following conditions:

1. An Indexed DD is absent and the associated mom DD is absent (Indexed DD present=0 and Memory [Indexed DD address] present=0);
2. An Indexed DD is dynamically generated from an absent DD and:

   a. The DD is a mom (DD present=0 and DD copy=0) or
   b. The DD is a copy and the associated mom DD is absent (DD present = 0 and DD copy = 1, and Memory [DD address] present = 0);

3. A segment Descriptor (SD) is absent (SD present=0).

A Presence Bit interrupt is not generated if the Descriptor is an absent copy, but the associated mom Descriptor is present. In that case the address field of the associated mom is used to make the required access.

Presence Bit interrupts generated for absent Data Descriptors will return as the P2 parameter either an absent copy data Descriptor (Indexed DD or DD), or a single-precision integer. Presence bit interrupts generated for absent code segment Descriptors will return as the P2 parameter an item with a TAG of 3, a Presence bit of 0, a Copy bit of 1, and an address field containing the address of the absent code segment Descriptor.

P2 parameter
: Action required by MCP

data desc
: Make the referenced array present

segment desc
: Make the referenced code segment present

sp integer
: For n=integer value: examine n stack items immediately below the interrupt MSCW, and for each item which is an absent Data Descriptor, make the referenced array present and replace the absent Data Descriptor by the present Data Descriptor (n counts 1 for a double-precision item)

The RCW created by an ODI interrupt entry normally points to the operator generating the interrupt. An exception occurs for Presence Bit on a code segment Descriptor generated by the stack structure operators ENTR, EXIT, RETN, and by dynamic branches. In these cases the RCW points to the code stream entry point being entered or returned to.

When an absent stack vector or stack Descriptor is encountered, a Presence Bit interrupt is generated with P2 holding the absent Descriptor and PbitAction = Exit. In most cases of SIRW evaluation, the operator may

be re-executed from its initial state. In other cases (for example, VALC) and in updating the stack addressing environment, re-executing the operator in restart state is sufficient to correctly implement the operator/MCP Presence Bit interface.

Paged Array

Paged Array is used by operators which process data arrays to indicate an attempted access beyond the end of the array. Operators which access a data array sequentially rely on the following assumptions:

1. A logical data array exists in memory as a set of 1 or more physical array pages managed by the MCP;
2. The elements of an array page are operands;
3. The next sequential word in memory after the last element of an array page has an odd tag.

Sequential processing of data arrays may be performed by pointer or edit operators. A Paged Array interrupt is generated by these operators when an odd-tagged item is read from an array or a store is attempted into an array word containing an odd-tagged item. If the odd-tagged access marks the end of the logical array, an error condition exists, and the operator cannot be resumed. If it marks the end of an array page but not the end of the logical array, the operator is resumed on the next page of the array.

Access to data arrays is through Indexed DDs (Indexed Word DDs or Pointers). An Indexed DD pointing to the "next element" of the array, the element whose attempted access produced an odd-tagged item, is the interface between the operator and the MCP interrupt procedure. Paged array interrupts return as the P2 parameter a single-precision integer, and the following MCP action is required: For n = integer value, examine n stack items immediately below the interrupt MSCW, and for each item which is an Indexed DD referencing an odd-tagged word, replace it by an Indexed DD correctly referencing the next array element (n counts 1 for a double-precision item).

Since B 5900 operators may have restart entry points unique from their initial entry points, preservation of partial results through interrupt processing can be implemented. Therefore, only the fixing of the Indexed DD stated above is required by the MCP for Paged Array interrupts.

Stack Overflow

Stack Overflow indicates that a push onto the expression stack has caused the size of the stack to equal its limit. The interrupt is a request to the MCP to extend the array in memory for the stack, and all operators may be resumed subsequent to MCP Stack Overflow processing, under the assumption that the stack size has been extended.

A Stack Overflow interrupt is generated when a word is pushed onto the expression stack, and at completion of the push, the address of the top of stack word (or, in the case of a push of a double-precision operand, the address of either the top-of-stack or second from the top-of-stack word) is equal to LOSR (Stack Descriptor address + Stack Descriptor length – 1). If a Stack Overflow condition occurs on pushing the first word of a double-precision item, the second word is pushed before the interrupt is generated.

For the interrupt entry sequence to complete without overwriting a value in memory, the array for the stack must include at least five words beyond the limit addressed by LOSR. Stack Overflow is not generated when a push completes with the top-of-stack address strictly greater than LOSR.

ERROR REPORTING

This set of interrupts reports error conditions arising from programming, compiler or operator faults. For all such interrupts if ID retry=1, operator arguments consistent with required initial state remain in the stack. In all cases the RCW created by the interrupt entry will point to the operator generating the interrupt.

Invalid Operator

An Invalid Operator interrupt is unconditionally generated by execution of NVLD (Invalid Operator). No other operator generates this interrupt. Below are definitions of some interrupt cases.

1. Attempted execution of undefined encoding -> Undefined Operator
2. Stack arguments of incorrect type -> Invalid Stack Argument
3. Invalid stack argument value -> Invalid Argument Value
4. Invalid code parameter value -> Invalid Code Parameter
5. Invalid indirect reference sequence -> Invalid Reference Chain

B 5900 generates Invalid Operator only for execution of NVLD.

Undefined Operator

An Undefined Operator interrupt is generated for the attempted execution of an undefined operator encoding.

Invalid Stack Argument

An Invalid Stack Argument interrupt indicates an invalid initial stack state for an operator. It will be generated by any operator which places data type restrictions on its dynamic stack arguments if one or more items on top of the stack do not have the required type(s). Argument type restrictions are in terms of the data types defined in Section 2 according to tag value and, in some cases, additional type bits within the word.

For all Invalid Stack Argument interrupts, the stack item which violates type restriction is passed as the P2 parameter. If two or more items are of incorrect type, only one is passed as P2. If the incorrect item is double-precision, the first word is passed as a single-precision operand with ID P2 double=1.

Invalid Argument Value

An Invalid Argument Value interrupt indicates that the data type of a dynamic stack argument is correct, but its value is not within a valid range. It will be generated if an operand argument interpreted as an integer produces an invalid value or if a field of a structured type has an undefined or invalid value.

The stack item having an invalid value is passed as the P2 parameter. If that item is double-precision, the first word is passed as a single-precision operand with ID P2 double=1.

Invalid Code Parameter

An Invalid Code Parameter interrupt indicates that a code stream parameter has an invalid value. It will be generated if a parameter interpreted as an integer produces a value greater than the maximum valid value. The invalid value is passed as the P2 parameter in the form of a single-precision integer.

Invalid Reference

An Invalid Reference interrupt indicates an attempted evaluation of an invalid address couple reference to an item in the current addressing environment. It is generated during evaluation of an NIRW or an address couple parameter under the following conditions:

1. The Lambda (lexical level) component is greater than LL (the lexical level at which the processor is running),
2. For Lambda=LL, the address of the referenced stack location is greater than the address of the top-of-stack.

If the invalid reference is an NIRW, the NIRW is passed as the P2 parameter. If the invalid reference is an address couple parameter, the P2 parameter is a single-precision operand whose low-order field is the address couple.

Invalid Reference Chain

An Invalid Reference Chain interrupt indicates that evaluation of an indirect reference produced an unexpected result. It is generated by operators which evaluate chains of one or more references if evaluation of an address couple parameter, NIRW, SIRW, or Indexed Word DD produces an item which is neither a valid reference in the chain nor a valid target item which terminates the chain.

The definition of valid target items varies according to the function of the operator. If an operator performs only a single reference evaluation, an Invalid Reference Chain interrupt is generated if the result is not a target. If an operator evaluates a sequence of references, the definition of valid reference chains also varies according to the function of the operator. In this case, an Invalid Reference Chain interrupt is generated if evaluation of a given reference is not a target or another valid reference in that context.

The invalid reference evaluation result is passed as the P2 parameter. If that item is a double-precision operand, the first word is passed as a single-precision operand with ID P2 double=1.

Invalid Index

An Invalid Index interrupt is generated if an integer value used to index a logical array of elements is not within a valid index range for that array. Invalid Index conditions may exist for indexing data Descriptors, code segment Descriptors, the stack vector Descriptor, or linear record structures by the operator OCRX. The P2 parameter varies depending on the type of array and the form of the index as noted in the following cases.

Data Descriptor (DD)

Invalid Index is generated if the index value is not in the range 0 to DD length–1, or if, in indexing a string DD or updating a Pointer, the computed word index is not in the range 0 to 2^{15}. For normal DD indexing cases, a copy of the DD is passed as the P2 parameter. For Pointer updating by pointer and edit operators, the P2 parameter is a copy of the Pointer, where the word index field is the computed index modulo 2^{16}.

Code Segment Descriptor (SD)

Invalid Index is generated by branching operators if the program word index component is not in the range 0 to SD seg length–1. If the new code stream pointer is specified by a PCW (dynamic branches, ENTR), the PCW is passed as the P2 parameter. If branching within the current code segment is indicated by a top-of-stack operand (dynamic branches), P2 is the operand, and if indicated by a parameter (static branches), P2 is a single-precision operand, where the low-order field is the parameter.

Stack Vector Descriptor (SVD)

Invalid Index is generated during stack accessing if the stack number is not in the range 0 to SVD length–1. An indexed copy of the SVD, where the index field is the invalid stack number, is passed as the P2 parameter.

Linear Record Structure

Invalid Index is generated by the operator OCRX if the sequence index operand is not in the range 1 to ICW sequence size. The single-precision ICW is passed as the P2 parameter.

Dynamic Branch Operand

If a dynamic branch within the current code segment is indicated by an operand on top of the stack and if the operand, after rounding if necessary, is not in the range (0 to 2^{13}), an Invalid Index interrupt is generated. The integerized operand is passed as the P2 parameter.

Memory Protect

A Memory Protect interrupt indicates an invalid attempt to write into a memory location. It is generated under the following conditions:

1. A write is attempted by store, overwrite, pointer, or edit operators, where where the memory location is referenced by an Indexed DD marked read only. The Indexed DD is passed as the P2 parameter;
2. Store operators encounter a tag 3 item in evaluating a reference chain or in writing a double-precision item, the second word location contains an odd-tagged item. The tag 3 or odd-tagged item is passed as the P2 parameter.

Divide by Zero

A Divide by Zero interrupt is generated by arithmetic divide operators if the numeric interpretation of the top-of-stack operand (the divisor) yields a value of 0.

Exponent Underflow

An Exponent Underflow interrupt is generated by arithmetic and numeric type transfer operators if the result of a rounding, truncation, or normalization function is too small to be represented by the machine.

Exponent Overflow

An Exponent Overflow interrupt is generated by arithmetic and numeric type transfer operators if the result of a rounding, truncation, or normalization function is too large to be represented by the machine.

Integer Overflow

An Integer Overflow interrupt indicates that an operand required to have an integer value cannot be represented as an integer. It is generated if the integer numeric value of the operand, after truncation or rounding if necessary, is not in the range (-2^{4} to 2^{38}) for single-precision or (-2^{79} to 2^{77}) for double-precision.

The operand is passed as the P2 parameter. If it is double-precision, the first word is passed as a single-precision operand, with ID P2 double=1.

Stack Underflow

Stack Underflow indicates that an operator attempted to pop an argument from an empty expression stack. The expression stack is the set of locations whose addresses are in the range (D[LL]+1 to LOSR), and a Stack Underflow interrupt is generated if the address of the top-of-stack is less than D[LL]+1 when a pop is attempted.

Bottom-Of-Stack

A Bottom-of-Stack interrupt indicates an attempted exit from the last activation record in the stack. It is generated by EXIT or RETN if at operator entry, D[LL] <= BOSR+1. The word at the D[LL] stack location is passed as the P2 parameter.

Stack Structure Error

A Stack Structure Error interrupt indicates an invalid condition in the stack linkage structures used to control procedure entry, procedure exit, and move stack operations.

The operators ENTR (including invocation on accidental entry), EXIT, RETN, and MVST will generate Stack Structure Error interrupts under any of the following conditions:

Stack [ADDR]
the stack item addressed by ADDR;

History Link
the address computed from a History Link;

Lexical Link
the address computed from a Lexical Link.

The item passed as the P2 parameter depends on the error condition, as noted.

If (stack [D[i]], stack [History-Link], or Stack [Lexical-Link] is not equal to a MSCW); or if (ENTR: Stack [F] is not equal to the inactive MSCW); or if (EXIT,RETN: Stack D[LL]+1 is not equal to the RCW); or if (MVST: Stack[base] is not equal to the TSCW), then P2=invalid stack item.

If EXIT,RETN: RCW ll or MVST: TSCW ll is not equal to MSCW ll, for the first entered MSCW on the historical chain whose head is the History Link corresponding to the RCW or derived from the TSCW, then RCW or TSCW.

If (History-Link is less than the BOSR value, then P2=MSCW containing the History Link); or if MVST: Computed F address is less than the value of BOSR, then P2=F address); or if (ENTR: top of stack address is less than F, then P2=TOS address; or if PCW ll is greater than 15, then P2=PCW).

For all Stack Structure Errors, the RCW created by the interrupt entry will point to the operator generating the interrupt.

Code Segment Error

A Code Segment Error interrupt indicates that in distributing a PCW or RCW code stream pointer, an invalid code segment Descriptor is accessed. It is generated if the tag of the stack item accessed at D[sdll]+sdi is not 3, where sdll,sdi are the code stream pointer components of a PCW or RCW. The invalid item is passed as the P2 parameter.

Invalid Program Word

An Invalid Program Word interrupt indicates that a word accessed from the current code segment is not a Program Code Word. It is generated in Table Edit mode if the tag of the word is not 0 or 3 and in all other modes if the tag is not 3.

The invalid item is passed as the P2 parameter. The interrupt RCW will point to the first syllable of the invalid word if the last valid operator occupied the final syllable(s) of the preceding Program Code Word. It will point to an opcode syllable of the preceding Program Code Word if that operator required syllables from the invalid word.

Alarm Interrupts

Alarm interrupts are triggered by hardware fault detection, and the RCW created by the interrupt entry will point to the operator which was executing when the fault was detected.

The Alarm ID parameter identifies the type of interrupt, and indicates by the retry bit whether or not the interrupted operator may be retried. Retry is set if the stack state at the time of the interrupt is still consistent

with the required initial state for the operator. More than one fault condition may be reported by a single Alarm interrupt. However, combinations exclude Local and Global Memory Uncorrectable Errors occurring together, as they are mutually exclusive.

Alarm ID (int class = 2)

E mode bit [28:1]
Constant value 1

int class [26:3]
Constant value 2 = Alarm

retry [19:1]
If 1, the stack state is still consistent with the required state for operator retry. If 0, stack arguments have been consumed, and an attempt to retry will produce an error condition.

int type [4:5]
The type of Alarm interrupt composed of:

inv add [4:1]
1=Invalid Address

LM error [3:1]
1=Local Memory Uncorrectable Error

GM error [2:1]
1=Global Memory Uncorrectable Error

hardware error [1:1]
1=Hardware Error

loop timer [0:1]
1=Loop Timer

LOCAL MEMORY UNCORRECTABLE ERROR

The P2 parameter identifies the memory address and the nature of the error. For read data errors, the word read from memory is pushed onto the stack immediately below the interrupt MSCW. Single bit read data errors are corrected by hardware and will not be reported by an interrupt, unless correction is disabled.

P2 parameter:

LM error type [47:7]
The Local Memory error field composed of:

LM single bit [43:1]
1=single bit read data error

LM multi bit [42:1]
1=multiple bit read data error

addr PE [41:1]
1=address parity error

address [19:20]
The memory address for the local memory operation

GLOBAL MEMORY UNCORRECTABLE ERROR

The P2 parameter identifies the memory address and the nature of the error. For read data errors, the word read from memory is pushed onto the stack immediately below the interrupt MSCW. Single bit read data errors are corrected by hardware and will not be reported by an interrupt, unless correction is disabled.

P2 parameter:

GM error type [47:7]
The Global Memory error field composed of:

write single [47:1]
1=single bit write transmission error

write multi [46:1]
1=multiple bit write transmission error

GM single bit [45:1]
1=single bit read transmission error

GM multi bit [44:1]
1=multiple bit read transmission error

read single [43:1]
1=single bit read data error

read multi [42:1]
1=multiple bit read data error

addr PE [41:1]
1=address parity error

address [19:20]
The memory address

LOOP TIMER

This interrupt indicates an effectively infinite loop by an operator. It is triggered by expiration of a timer whose interval is sufficient for valid execution of any operator. A Loop Timer interrupt indicates an operator fault except for LLLU (linked list lookup), which may encounter a data-driven, non-terminating loop.

HARDWARE ERROR

This interrupt indicates a hardware detected error which is uncorrectable.

[31:32] CPU Detected Parity Error
- Bit0 = DP1 MBUS
- Bit1 = DP2 MBUS
- Bit2 = HDP1 MBUS
- Bit3 = IT1 MBUS
- Bit4 = MC1A MBUS
- Bit5 = MC2A MBUS
- Bit6 = MC1B MBUS
- Bit7 = MC2B MBUS
- Bit8 = PC1 MBUS
- Bit9 = HGMI MBUS
- Bit10 = SLC5 MBUS
- Bit11 = IT2 MBUS
- Bit12 = DP3 PROM
- Bit13 = DP4 CBUS or PROM
- Bit14 = DP1 CBUS or PROM
- Bit15 = GND
- Bit16 = DP3 MBUS
- Bit17 = DP4 MBUS
- Bit18 = GND
- Bit19 = HARDERR (Bad Mem Cntl)
- Bit20 = SLC1 RAM
- Bit21 = SLC2 RAM
- Bit22 = SLC4 RAM or PROM
- Bit23 = SLC5 RAM
- Bit24 = SOFTERR (Stk Over)
- Bit25 = HDP2 CBUS or PROM
- Bit26 = IT1 CBUS
- Bit27 = MC2A CBUS
- Bit28 = MC2B CBUS
- Bit29 = PC1 PROM
- Bit30 = PC2 CBUS
- Bit31 = HGMI CBUS

External Interrupts

External interrupts are invoked between operators to report events which are independent of the executing code stream. The RCW created by the interrupt entry will point to the next operator in the interrupted code stream.

The External ID parameter identifies the type of interrupt. More than one external event may be reported by a single External interrupt.

External ID (int class = 4)

E mode bit [28:1]
Constant value 1

int class [26:3]
Constant value 4 = External

int type [3:4]
The type of External Interrupt composed of:

global alarm [3:1]
1=Global Alarm

IO [2:1]
1=I/O Finished

global [1:1]
1=Global Attention

int timer [0:1]
1=Interval Timer

External interrupt are masked by the CS (control state) flip-flop except for the Global Alarm.

SECTION 5
SYSTEM CONCEPT

FUNCTIONAL DESCRIPTION

The various modules of the Central Processing Cabinet are discussed individually in the following paragraphs.

DATA PROCESSOR

The Data Processor is a four card module occupying the number 6 to 9 slots of panel A in the B 5900 backplane. The Data Processor performs all arithmetic and logical operations in the B 5900. The machine state is contained in 16 registers in the AMD2901B processors used in the Data Processor. The DP consists of these submodules.

Arithmetic Logic Unit and Register Stack (Data Path).
Literal and Field Isolation Unit.
Tag and lexical level Controller
Latch
Control Logic

The Data Processor is two major modules that communicate with the other modules of the B 5900 by way of the M-bus. The Stored Logic Controller is hardwired to the DP. The SLC controls ALU and Stack functions by direct control signals. Operations by the ALU and Stack are performed independently of the data transfer taking place on the M-bus. The internal data path in the 2901 Processor chip provides this capability. The Isolation unit is controlled by a command field of the C-bus. The input to the Isolation unit is entered on the M-bus and the result is placed on the D-bus. Therefore, arithmetic or logic operations with data in the stack or literal data do not require C-bus or M-bus resources, while a field isolation operation requires the full machine bus capabilities.

ALU AND REGISTER STACK

The ALU and Register Stack consists of 12 AMD2901 4-bit slice processors. This module contains a 48-bit ALU capable of eight binary, logical, and arithmetic operations, sixteen 48-bit registers configured as a two port, two address RAM, a 48-bit sidecar register (Q register) capable of storing and shifting temporary data, and a variety of multiplexors and tri-state drivers which enable the module to interface directly to the M-bus. In addition to the AMD2901s there are four carry-lookahead units that improve arithmetic performance. A decimal adjust function is attached to the AMD2901 inputs. The decimal adjust logic consists of an auxiliary carry register and control gates. When the decimal adjust line is one, subtraction of the B input from the A input of the ALU is suppressed if the auxiliary carry for that slice is one. These 16 chips form the entire arithmetic section of the B 5900 processor.

Figure 5-1 is a functional block diagram of the B 5900 Data Processor module.

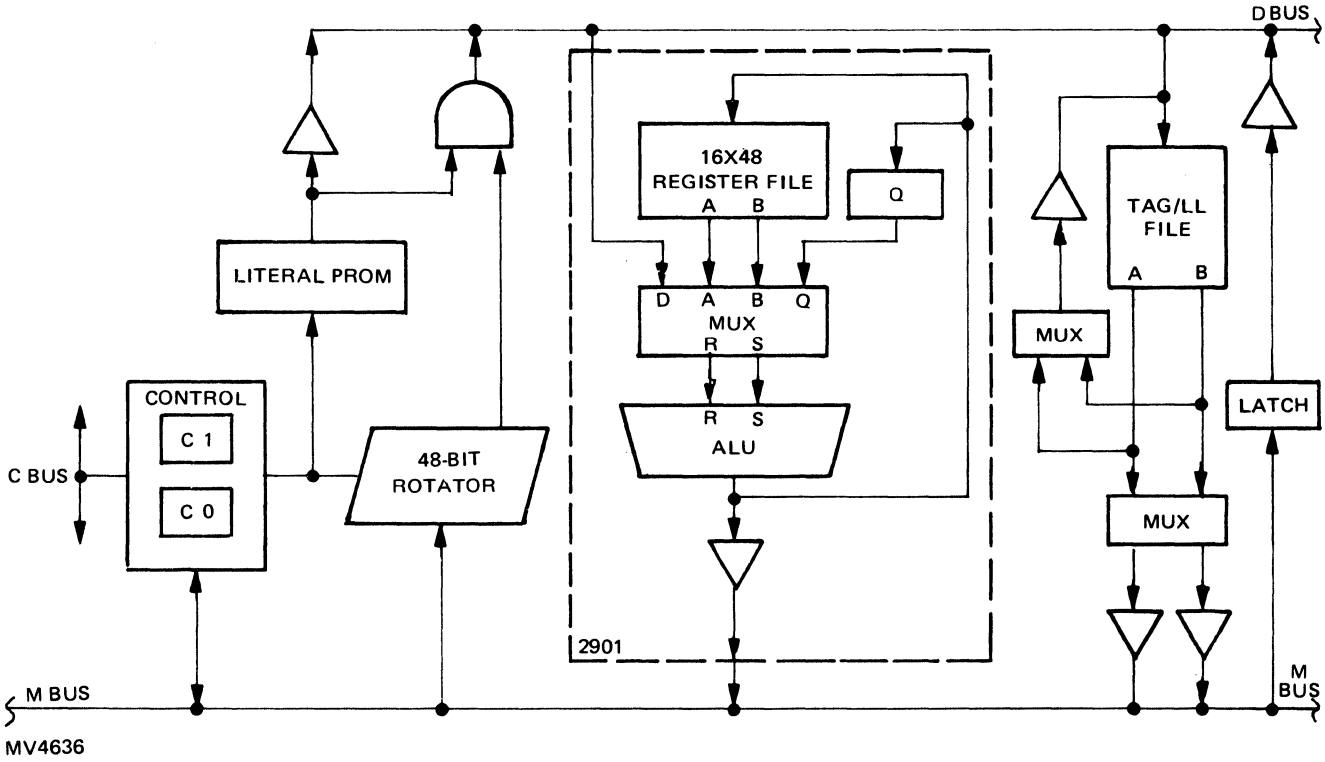


Figure 5-1. Functional Block Diagram of the Data Processor

ROTATION AND FIELD ISOLATION UNIT

This module consists of a 48 bit rotator and masker. The module can perform any rotation from 0 to 48 bits, and can apply data from the literal PROM to logically AND the rotated result. The rotation is End Around and is controlled by the dynamic address control logic. The output of the Isolation unit is placed on the D-bus and returned to the 2901 at the D inputs (Data in).

LITERAL PROM

The Literal PROM is a 1 K by 48-bit PROM that contains patterns which are used as constants in ALU calculations, or as input to the field isolation portion of the rotation unit. The Literal PROM may be addressed dynamically or statically from the C-bus. The literal data is placed on the D-bus and is used in both arithmetic and logical operations in the 2901 ALU. All the Literal PROM addressing is controlled by the dynamic address controller.

ROTATOR

The Rotator consists of 48 4-bit shifters arranged in a 3 level rotation pattern. The Rotator implementation is that of a truncated 64-bit barrel shifter, and is controlled by eight control lines. These eight lines cause a left shift of from 1 to 48 positions. Because of the nature of the shifter, some rotations are duplicated, that is, two different 8-bit patterns result in the same rotation. The 8-bit number supplied to control the rotator is not related to the number of bits rotated. In the Dynamic Address Controller, a PROM (the rotate PROM) is used to map binary numbers from 0 to 47 into the proper rotation 8-bit number.

DYNAMIC ADDRESS CONTROLLER

The Dynamic Address Controller commands the rotation and literal PROM units. The control lines for the Rotator and Literal PROM originate from the SLC (static control) or from a combination of SLC static data, and the contents of counter registers in the Dynamic Address Controller. The Dynamic Address Controller contains two 8-bit counters that are loaded from, or returned to the M-bus under SLC control. The counters address the Rotator, the Literal PROM, or both simultaneously. The counters are incremented or decremented at the end of the cycle by the SLC. The SLC controls multiplexors that allow selection of combinations of counters and SLC bits when forming Literal PROM and Rotation addresses. The rotation address is passed through a Mapping PROM prior to being applied to the Rotator. This allows the mapping out of the hardware dependent rotation addresses (see Rotator heading) and allows the interpretation of addresses in different ways; that is, a one in a counter can be interpreted as a one bit, or one byte, rotation, depending on certain static SLC bits on the Rotate PROM address.

TAG AND LEXICAL LEVEL CONTROLLER

In order to facilitate the handling of tags and lexical level data, this module contains both storage for tags and lexical levels, and logic to perform lexical level address computation. The storage section consists of 16 4-bit words, one for each register in the stack. These words may contain a tag or lexical level depending on the contents of the related word in the stack. The B 5900 supports four resident display registers (D registers) in the hardware. These registers reside in the stack. The D registers supported are D[0], D[1], D[2], and D[LL].

D[0] through D[2] must reside in register stack addresses 0 to 2. D[LL] may reside in any of the other registers. D[LL] is the register in which the lexical level that the machine is running is displayed. In an address computation, the processor must add an offset (Delta) to one of the D registers. The proper register is selected by a Lambda value coming from the Program Controller. The lexical level Controller is instructed to obtain the Lambda-Delta pair from the Program Controller, decode it on the first clock of an address computation sequence and place the Delta value on the D-bus along with selecting the proper D register to be added to the Delta value. The data path ALU is used to perform the addition required to generate an address on the second (and final) clock of an address computation sequence.

The tag and lexical level control is also capable of placing a tag on the D-bus in bit positions [3:4] and reading tags in that position from the M-bus. The tag register is a two port register. It is possible to command the controller to place one tag in M-bus position [50:3] and another on condition lines to the SLC to allow dual tag comparison in the SLC. Figure 5-2 is a block diagram of the Tag and lexical level Controller.

LATCH

The Data Processor contains a latch function that allows asynchronous requestors to deposit data for use by the DP, and is a 51-bit latch which serves as an interface between the system M-bus and the internal D-bus (51-bit) in the DP. The latch is loaded asynchronously by a requestor or under SLC control using the LATCH control bit. The Latch is capable of operating in one of two modes. If the LATCH bit is 1 or a MHLD signal is present, the LATCH is open and its outputs will follow its inputs. When the MHLD signal is removed, or the cycle during which the LATCH bit was on ends, the latch closes, holding the data at its inputs. This data will remain in the latch until another LATCH command or MHLD command is received. The output of the latch is connected to the D-bus. The outputs can be in one of two modes. When the latch is open (LATCH is 1 or MHLD) the output drivers are enabled, the outputs of the LITERAL and ISOLATE units are disabled automatically. When the latch is closed, its outputs are enabled on cycles where the LITERAL and ISOLATE units are not being used (that is, OP DEF not 0). When LITERAL or ISOLATE units are used, the DMOD field allows D DATA or D TAG to be modified, or when an LDREAD is requested, the latch outputs are disabled.

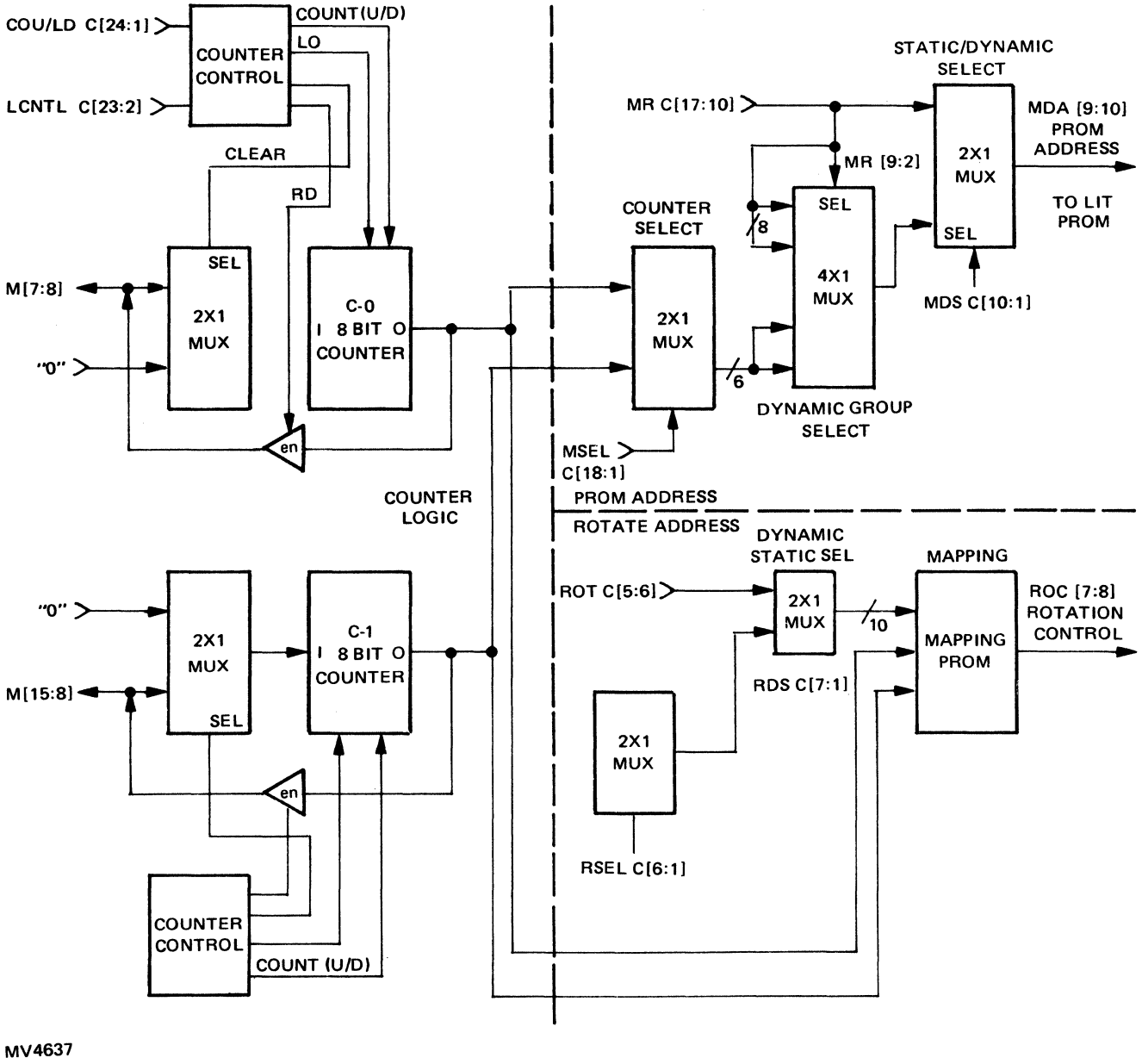


Figure 5-2. Functional Block Diagram of the Tag and Lexical Level Controller

D-BUS

The D-bus is the internal data bus of the DP. The DP uses the D-bus to obtain all input data to the 2901 data path and the tag registers. There are several D-bus sources, these include:

Tags

These are placed on the D-bus in bit position [3:4] by the tag file. They are controlled by the DMOD field of the SLC and have the highest priority on the bus in case of conflict.

Lambda and Delta
: The output of the lex level controller which can be read by the SLC command LDREAD, or placed on the D-bus automatically during the second clock of an ADTRAN sequence. This operation has second priority in case of bus conflict.

Rotation and Literals
: The output of the Isolation unit, or the Literal PROM. These values are placed on the D-bus whenever the DP is selected on the C-bus and one of the above sources is not enabled.

Latch
: The latch is the primary means of sending data to the DP registers from other parts of the machine. The latch outputs have highest priority when the latch is open. When closed, it is the default source on the D-bus when none of the above sources are enabled.

The D-bus is entirely internal to the DP and is not shared with any other module. It is parity checked when it is connected to the M-bus through the latch.

CONTROL LOGIC

The Control Logic decodes commands on the C-bus and generates the proper signals to control the submodules. The Control Logic samples the OP field of the micro control word every machine cycle, and if the DP is being addressed, generates the proper enable signals.

BUS CONTROLLER

The Bus Controller ensures the proper CPU module (PC, DP, MLIP, MP, or MC) can gain control of the M-bus. There are two methods of gaining control of the M-bus to transfer data:

Programmed Control
: Control of the bus can be assigned to a specific module under control of the SLC. The SLC microword contains a 3-bit field that defines only one module to be enabled on the bus.

Asynchronous Control
: Control of the bus can be gained by memory modules for transfer of data into the Latch. This transfer is transparent to the SLC. The other modules in the machine are frozen by the MP which gates the system clock for the duration of the transfer. One HOLD signal is generated by wire-ORing the outputs of the memory controls. Selection of the unit to be enabled is performed by scanning the INVADRn status lines from the async (memory) requestors.

The bus control consists of two major modules:

The Program Bus Controller

The program bus controller enables the tri-state controls of the module indicated by the mode lines from the SLC. The program bus controller can be disabled by an Async request. The following signals are inputs to the program bus controller.

BUS(3)
: Bus select lines from the SLC define which tri-state outputs to enable. The selections available are;

DP	0	SPARE	4
PC	1	I/T	5
M(L or G)	2	SPARE	6
MLIP	3	Adapters	7

Asynchronous Bus Request Controller

The Asynchronous bus controller accepts a HOLD signal and generates the proper tri-state enable signal to the proper MC. The controller also generates a signal to hold the program bus controller. The input is MHLD/. Up to three modules may be selected in async mode, depending on the state of the INVADRn lines from the requesting module. The requestor must bring its MHLD line low prior to the clock during which it wishes to gain control of the bus. The module must hold MHLD TRUE until it is serviced. The highest priority module is serviced first. All other modules must hold their MHLD signal TRUE to be considered for service on the next clock.

DATA PROCESSOR MAINTENANCE MODE

There are two signals that control the data processors state in maintenance mode, these are the HCPMODE signal and the ABORT signal. There are three ways of accessing state in maintenance mode. These are shift registers, registers (and counters), and PROMs. In HCPMODE shift registers are placed under direct control of the MP and are read and altered. The registers remain frozen even with clocks running if not selected by the Maintenance Processor. Registers, such as the stack in the 2901, are controlled by the ABORT signal. This allows reading of register contents, but does not allow the registers to be updated. To update registers, microprogramming in the SLC must be present that specifies the desired register ONLY to be updated, all other states remain the same. In DISPLAY mode, both HCPMODE and ABORT are on, and all flip-flops in shift registers are readable and setable, while all registers are readable. In UPDATE mode, only HCPMODE is on, registers are set under control of the SLC. All shift registers in the data processor are cleared to logic 0 by the GCLR/ (General Clear) signal.

STORED LOGIC CONTROLLER

The Stored Logic Controller is a 5-card module that resides in the number 12 to 16 slots of the B 5900 Central Processor backplane.

The Stored Logic Controller (SLC) provides the Micro Control Word to execute in the rest of the machine as well as determines the next state of the machine. The SLC uses two types of cycles in determining the next state. The first cycle is the assumed cycle and every micro-statement begins with this cycle. In this cycle, the next state is assumed, even before conditions are received from the various modules of the machine. When the conditions become valid, two paths are available. If the conditions verify the assumed choice, then no change is made and the next cycle type is another assumed cycle of the next state. If the conditions contradict the assumed choice, the SLC enters the second cycle type; that is, the corrective cycle. In this cycle the correct state must be chosen according to the known conditions. A penalty of one clock cycle is paid whenever the corrective cycle is enacted. A corrective cycle is always followed by a preferred cycle of the correct next state.

The selected conditions also determine if the work being done in the present machine cycle should be aborted. If so, the abort signal from the SLC to the rest of the modules becomes active. A penalty of one clock cycle is paid for this abort of the present cycle. If the SLC enters a corrective cycle (aborting the next cycle)–in addition to the abort of the present cycle–only one clock cycle is paid for the abort. The SLC contains the following modules as shown in Figure 5-3.

Control Store
Sequence Store
Condition Select
State Select
Subroutine Stack

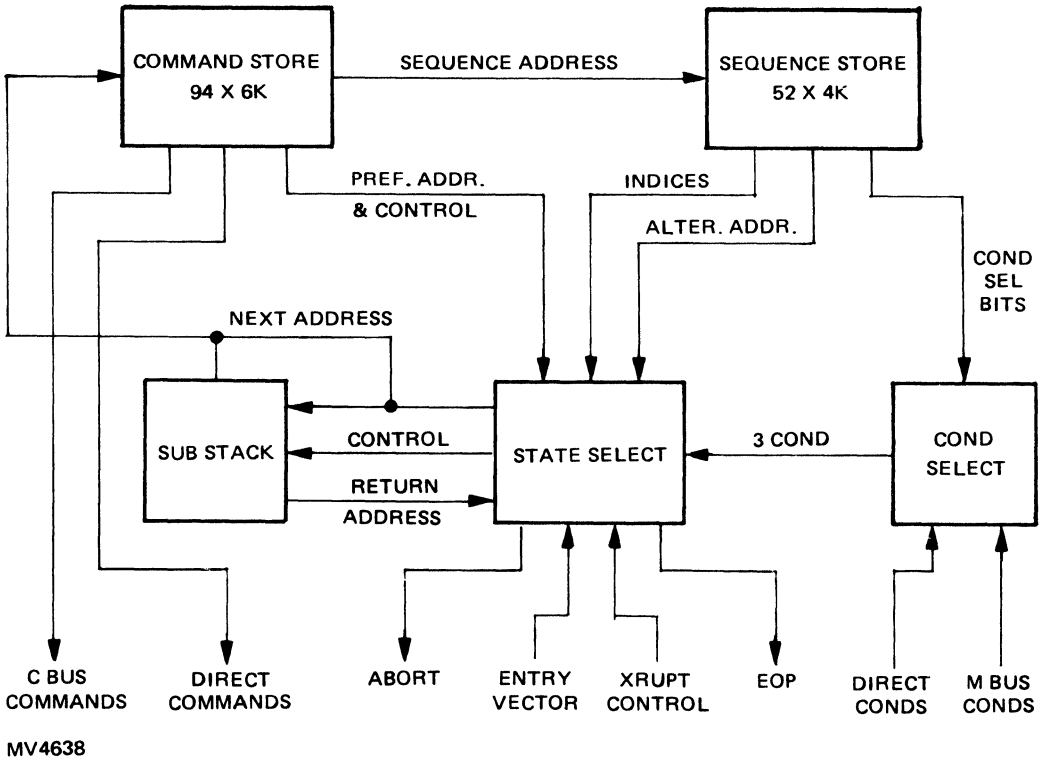


Figure 5-3. Stored Logic Controller Modular Block Diagram

CONTROL STORE

This module contains the Control Memory (CM) and its associated circuitry. The Micro Control Word and fields which are necessary for sequencing are housed in the CM in a 8k by 94 RAM matrix. This memory uses 4K by 1 static RAM chips. To achieve the 8K height, two layers of 4K by 94 are tri-stated together using chip selects with a thirteenth address bit. The module has buffers for the 13 address lines and a Command Register to staticize the RAM outputs. Additional hardware provides maintenance capabilities for the CM. Buffers to and from the M-bus provide a data path to write and read the memory in maintenance mode. Parity is checked separately in the three partitions of CM due to implementing the CM on three different SLC boards. SLC boards 1 and 2 are logically identical boards and are interchangeable. The difference is the interpretation of the fields implemented on the different boards by the system. Figure 5-4 is a functional block diagram of the Control Store.

There are three major fields in the CM. The concurrent part of the Micro Control Word basically is the direct control to the DP. The shared part of the Micro Control Word is the C-bus. The third field is used by the SLC to sequence through the microprogram. The individual fields and the boards they reside on are as follows:

SLC1

CM[93:32]
CM[93:01] Odd Parity for CM[93:32]
CM[92:03] M-bus Controller
CM[89:01] Latch Open
CM[88:15] Preferred Address
CM[73:01] Assumed Select
CM[72:11] Sequence Memory Address

SLC2

CM[61:32]
CM[61:01] Odd Parity for CM[61:32]
CM[60:31] Micro Control Word

SLC3

CM[29:30]
CM[29:01] C-bus parity
CM[28:03] C-bus address
CM[25:26] C-bus command

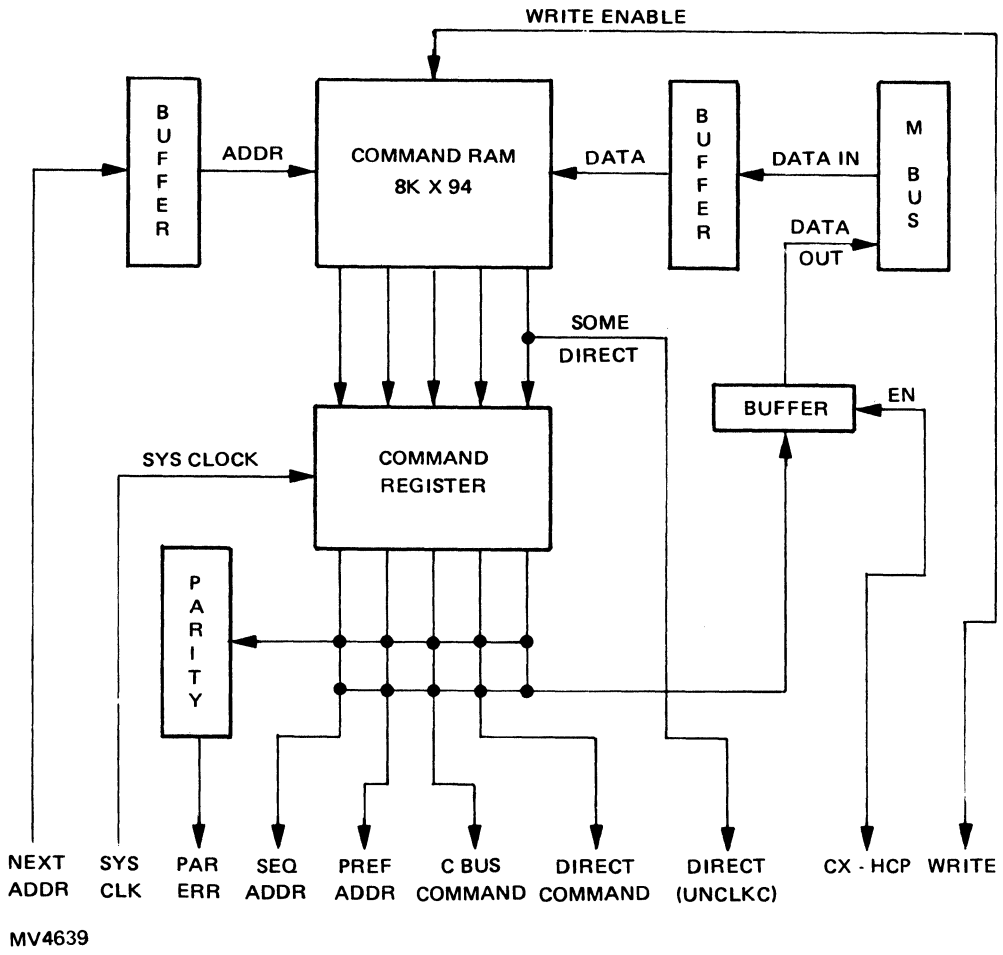


Figure 5-4. Control Store Block Diagram

SEQUENCE STORE

This module contains the Sequence Memory (SM) and its associated circuitry. The SM is a 4K by 52 RAM matrix of 4K by 1 static RAMs. As with the CM, the SM has buffers to and from the M-bus to provide data paths to read and write the memory in maintenance mode. The SM is housed on two boards, SLC4 and SLC5, along with logic from other SLC modules.

The fields of the sequence memory are the condition select field, the indices field, and the three alternate address fields. This memory is partitioned both horizontally and vertically to obtain these fields. The three alternate address fields are contained in the "upper half" of the sequence memory while the condition select and indices are contained in the "lower half". Because of this architecture, all fields of the sequence memory are not available at the same time. The lower half containing the indices and condition select fields are usable during a SLC assumed cycle. When a SLC corrective cycle is needed, the most significant bit of the sequence address is modified to make the upper half of the memory available. Since these alternate addresses are not needed in the preferred cycle, this allows for a significant decrease in memory width. The upper and lower halves of the sequence memory differ only in the most significant bit of address and that the control memory views the sequence memory as twice as long but only half as tall. The condition select field is used in the Condition Select module to eventually select the three desired conditions. The indices are used by the State Select module to determine the next address.

One location of the sequence memory is shared by all states of the Control Memory which have an unconditional next state. The fields in this location cause the SLC to neglect the condition logic and to take the next address dictated by the assumed select field of the CM.

The fields of the SM are defined as follows:

Sequence Memory Lower 2K (assumed cycle)

SM[51:01] Odd Parity for SM[51:33] (on SLC 4)
SM[50:01] Try7
SM[49:03] Index7
SM[46:01] Try6
SM[45:03] Index6
SM[42:01] Try5
SM[41:03] Index5
SM[38:01] Try4
SM[37:03] Index4
SM[34:01] Try3
SM[33:03] Index3
SM[30:01] Try2
SM[29:03] Index2
SM[26:01] Try1
SM[25:03] Index1
SM[22:01] Try0
SM[21:03] Index0
SM[18:01] Odd Parity for SM[18:19] (on SLC5)
SM[17:06] Condition Group 2 Select
SM[11:06] Condition Group 1 Select
SM[05:06] Condition Group 0 Select

Sequence Memory Upper 2K (corrective cycle interpretation)

SM[51:01] Odd Parity for SM[51:33] (on SLC4)
SM[50:02] Unused
SM[48:01] Alt Addr 3 Return Offset Bit
SM[47:01] Alt Addr 3 Enter Bit
SM[46:13] Alt Addr 3
SM[33:01] Alt Addr 2 Return Offset Bit
SM[32:01] Alt Addr 2 Enter Bit
SM[31:13] Alt Addr 2
SM[18:01] Odd Parity for SM[18:19] (on SLC5)
SM[17:03] Unused
SM[14:01] Alt Addr 1 Return Offset Bit
SM[13:01] Alt Addr 1 Enter Bit
SM[12:13] Alt Addr 1

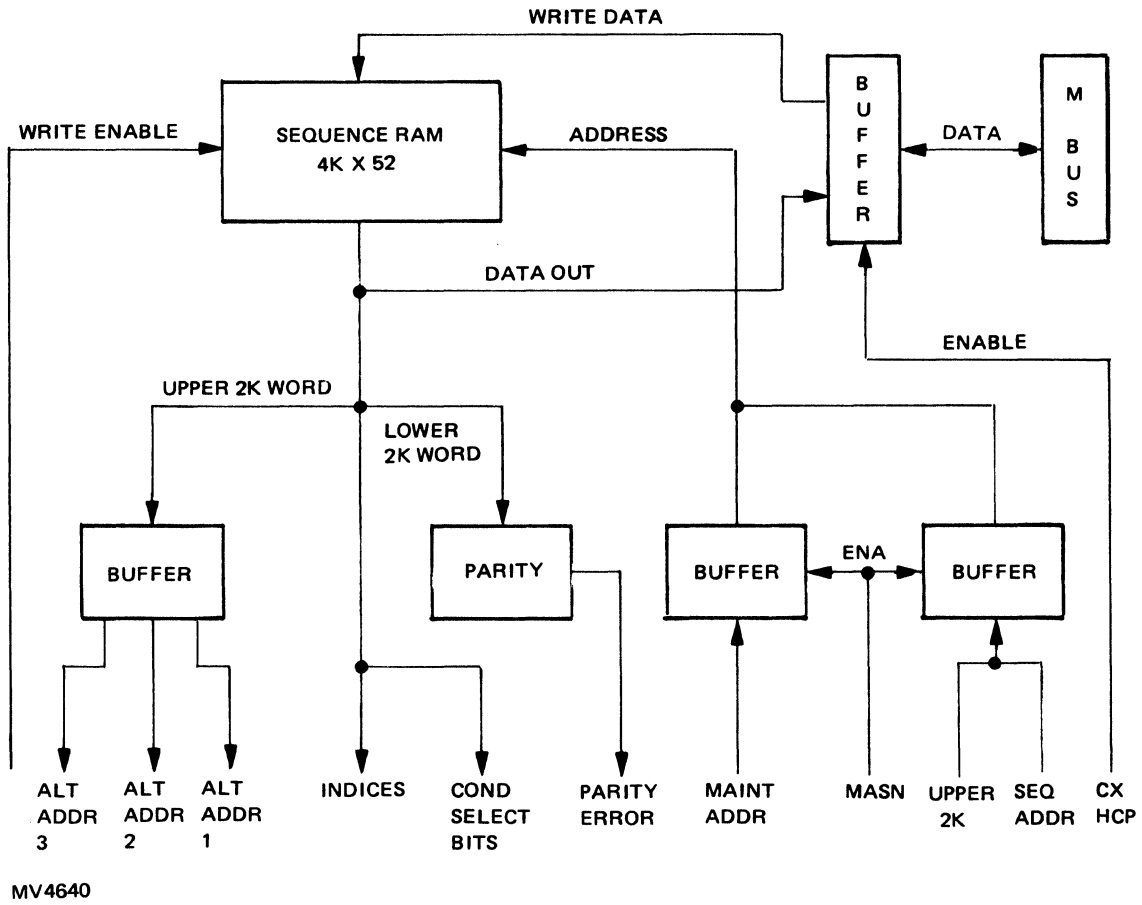


Figure 5-5. Sequence Store Block Diagram

CONDITION SELECT

A conditional microstatement can look at up to three conditions. The Condition Select Module has the B 5900 condition set divided into three groups. The Sequence Memory has three fields (condition select fields) each of which selects 1 out of 56 conditions for one condition group.

Some conditions are found only in one group and are called unique conditions. Groups were deliberately arranged because unique conditions of the same group cannot be tested in the same machine cycle. Some conditions are available to all three groups and are called common conditions. As long as the total number of conditions (three) is not violated, a common condition can always be tested since they do not have the grouping problems of unique conditions.

Some conditions are gated together to form more sophisticated conditions. This is accomplished through the use of gate arrays (Programmable Logic Arrays or PALS). The output from these arrays are referred to as encoded conditions. Encoded conditions are a function of up to ten input conditions but still count as one condition.

A few conditions become valid very late in the machine cycle and are stored and used in the next microinstruction. This means that if one of these late conditions is selected, the state of what occurred in the previous machine cycle is tested. Figure 5-6 illustrates the Condition Select module.

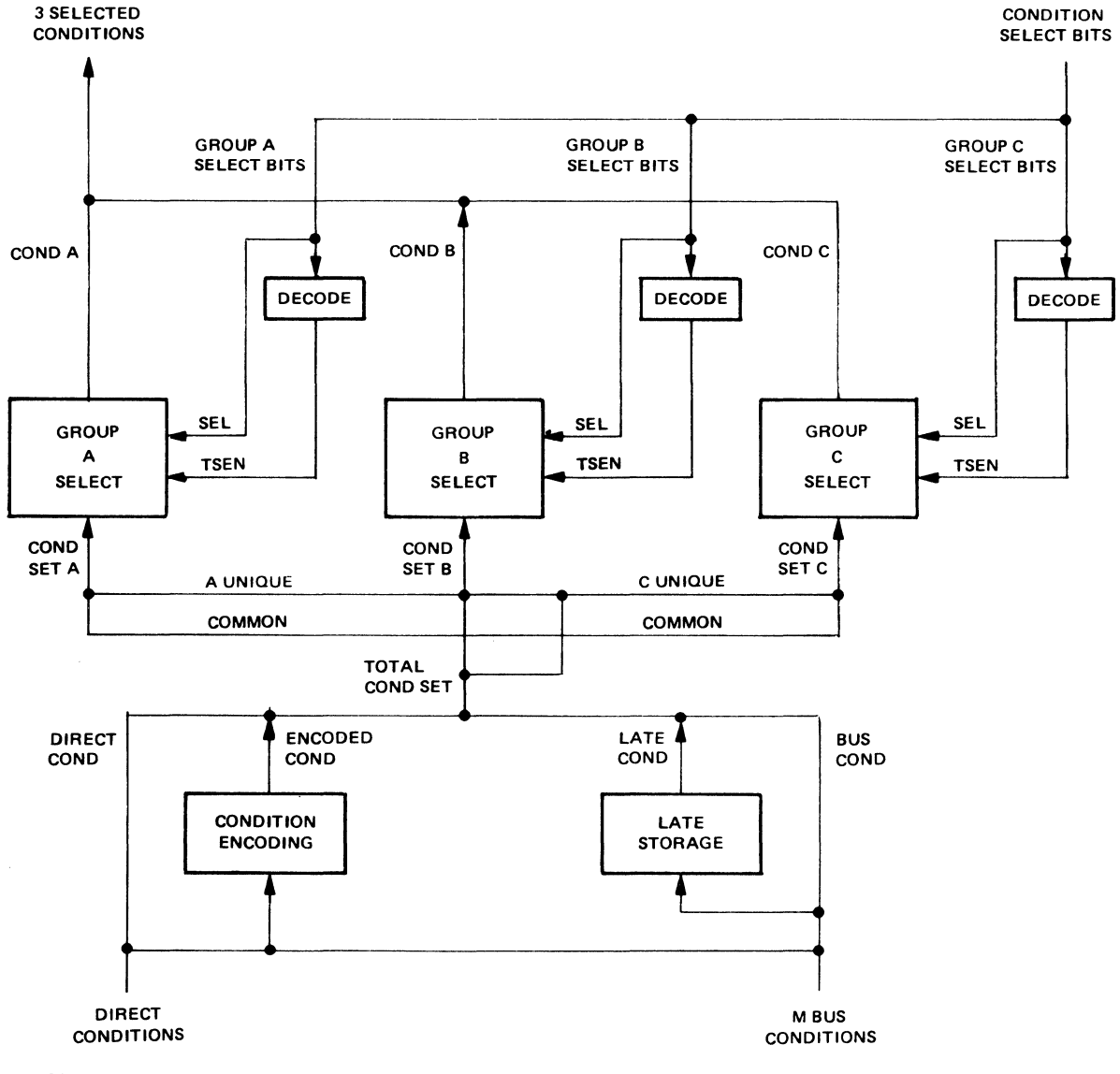


Figure 5-6. SLC Condition Select Block Diagram

STATE SELECT MODULE

This module determines the next address of the SLC microprogram. This module contains Index Selectors, an Index Register, an Address Select PROM, and Address Selectors.

There are eight possible next addresses for the SLC. These possible next addresses are: preferred address from CM, three alternate addresses from the SM, return address from subroutine stack, entry vector from the Program Controller, microinterrupt address, and an external interrupt address.

In an assumed cycle, the microprogram can only have the preferred address, the return address, or an entry vector as the next address. This selection is achieved in the PROM with assumed select line from the CM and the stack=0 line from the Subroutine Stack as addresses.

Assumed Select	SUBSTK Zero?		Next Address
0	x	→	Preferred
1	0	→	Entry Vector
1	1	→	Return

The preceding table shows that if the assumed select line is 0, the preferred address is selected. If the assumed select is 1, the state of the subroutine stack must be interrogated. If the stack is empty (0), this is interpreted as the end of the operator and an EOP (End of Operation) signal is generated from the PROM. If the stack is not empty, it is interpreted as a normal return from a subroutine and the return address is selected.

When the SLC is in a corrective cycle, the value of the Index Register overrides the assumed selection and selects the correct next address according to the following table.

Index	Cycle Type	Next Address
0	corrective	invalid case
1	corrective	preferred
2	corrective	return
3	corrective	unused
4	corrective	alternate 1
5	corrective	alternate 2
6	corrective	alternate 3
7	assumed	assumed choice

The Index column does not refer to the index number but to the binary value of the selected index. When the selected index equals 7, this tells the PROM that the assumed selection is correct and the next cycle type will be another assumed cycle. Any index value other than 7 causes a corrective cycle.

The module receives a group of signals from the SM which are the indices. These indices are arranged into 8 groups of 4 bits each. The 3 selected conditions from the Condition Select Module in turn select 1 of the 8 index groups. This selected 4-bit index has 3 bits of address select data and 1 bit of abort present cycle information. If the abort bit of the selected index is 0 the next system clock is masked to give the rest of the system time to abort the present cycle. If the address select portion of the index does not equal 7, the SLC aborts the next cycle by going into a corrective cycle. This also masks the clock because of the extra accessing time required in the CM. If the 4-bit selected index is all ones, then there are no aborts, no clocks are masked, the assumed address is the next address, and the next cycle is an assumed cycle.

Interrupt routines are entered through this module. Micro and external interrupt detection signals are sent by the Interrupt Logic module. These lines address the PROM. The PROM is coded to give microinterrupts highest priority. When this signal is active, the PROM gives address selection bits which select the address dedicated to appropriate routine. The external interrupt is similar except it is lower in priority than the microinterrupt and also has a mask signal which prevents the external interrupt from being serviced. This mask line comes from the Program Controller and is active during edit mode. The SLC also insures that external interrupts are serviced only between operators.

When the SLC goes into a corrective cycle, the most significant address bit of the sequence memory address field is forced to one. This makes the three alternate addresses available by going into the upper 2K of memory as explained in the Sequence Memory section. This also means that the condition select bits and the indices in the lower 2K become invalid. To circumvent this problem, the selected index is captured in a register which is clocked by an unmaskable clock.

The actual address selection is made by multiplexers which are selected by bits of the address select PROM. The outputs of these multiplexers are tri-stated with buffers of the maintenance address from the Maintenance Processor (MP). The MP also controls this tri-stating.

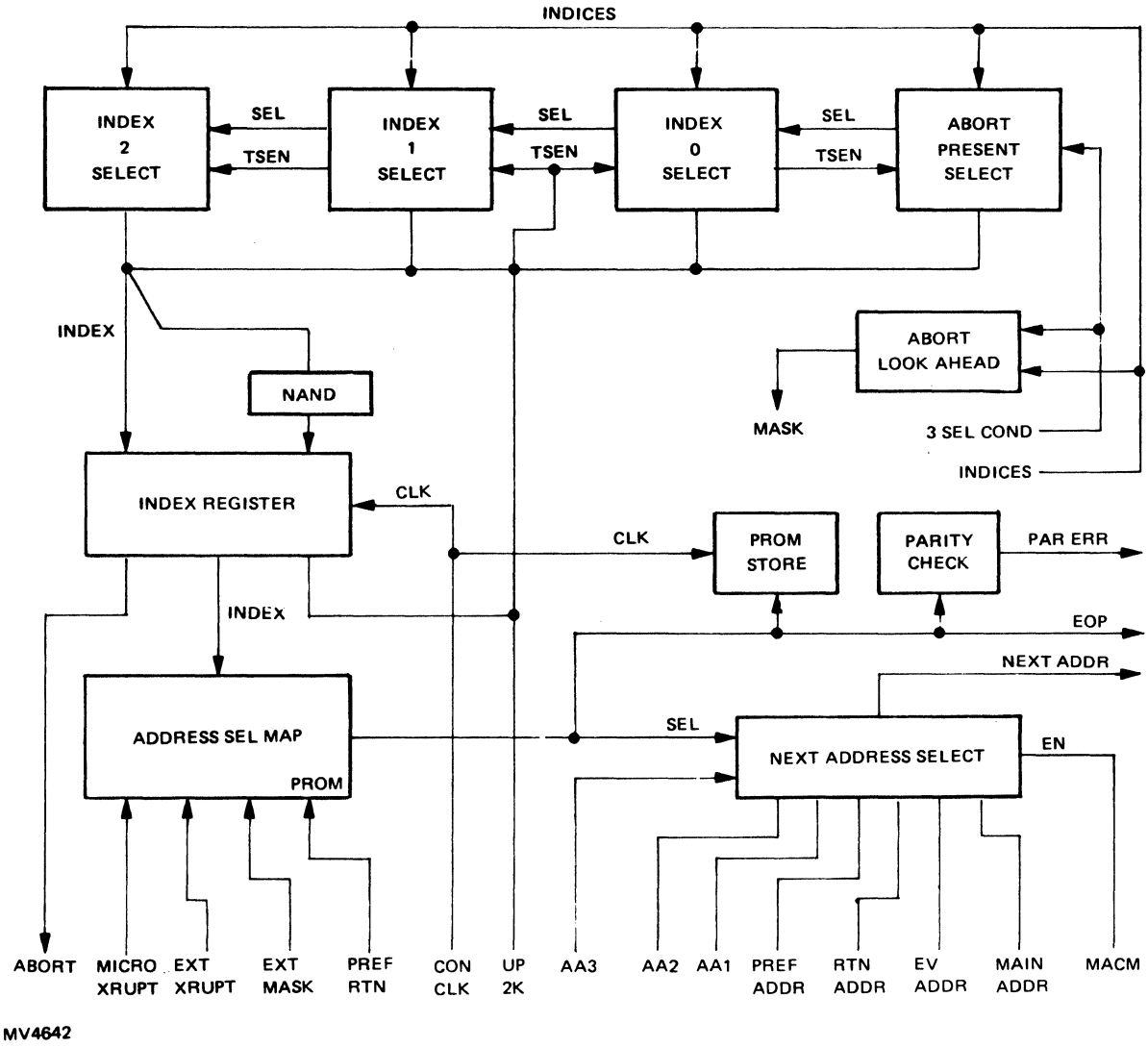


Figure 5-7. State Select Module Block Diagram

SUBROUTINE STACK

This module facilitates microprogram subroutining. The module is a last in – first out (LIFO) stack. The stack is pushed down with the modified present address to the control memory during states that call subroutines. States that call subroutines are identified by an extra bit added to the calling state address. If this bit is high, then a call state is identified. Another bit following the enter bit is a return address offset bit. This bit is added to the least significant 4 bits of the calling state address which becomes the return address in the stack. These two bits are used by the subroutine mechanism only, but are a part of the preferred address (from the control memory) and the three alternate addresses (from the sequence memory). Calling can occur in the preferred cycle or correction cycle, as can returning. In a return, the address at the top of the stack becomes the next address to the control memory. The stack is then popped. Provisions are made for handling 14 levels of subroutine nesting. Subroutine stack overflow error condition is reported when the stack capacity is exceeded.

No stack action is taken upon a return when the stack is empty. The SLC interprets this as an EOP. The stack can be conditionally flushed with the Program Controller C-bus interface.

PROGRAM CONTROLLER

The Program Controller provides facilities to map operators into entry vectors for the SLC and to extract and build parameter words from the code stream to transfer to the DP.

Upon receiving an EOP from the SLC, the PC begins working on the next operator. While the SLC executes the present operator, the PC fetches the next operator and gets the entry vector and parameter word associated with it. Figure 5-8 is the Program Controller block diagram.

The PC consists of the following modules:

Syllable Prep	Parameter Prep
Operator Map	Control

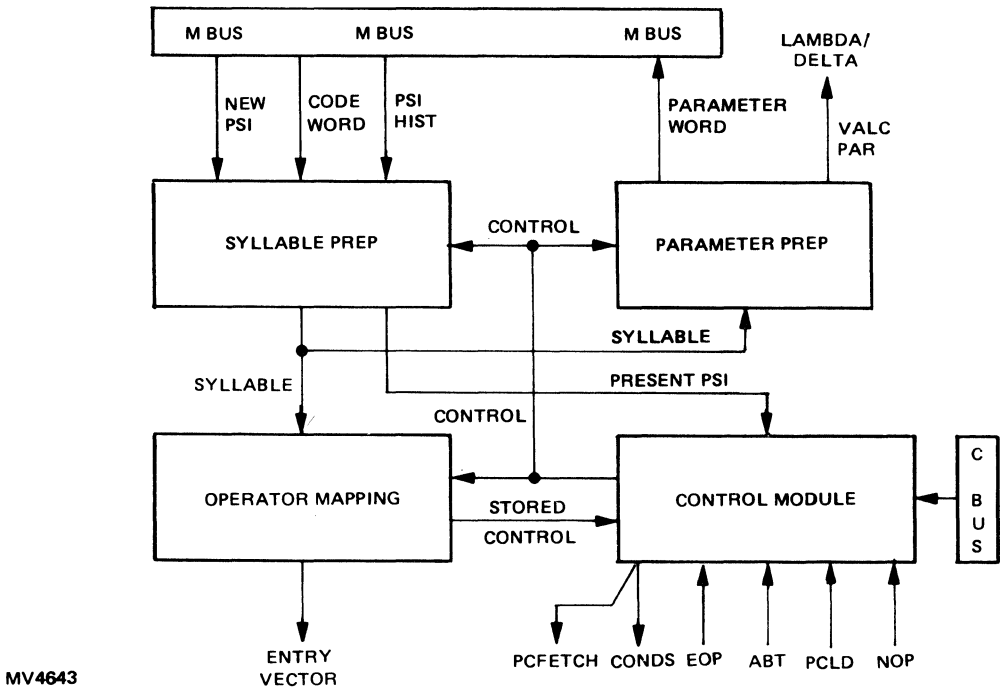


Figure 5-8. Functional Block Diagram of the Program Controller Module

SYLLABLE PREP MODULE

This module extracts one syllable at a time from program code words and keeps track of where in the code stream the PC is operating. To minimize the effect of word boundaries in the code stream, the PC can hold two program code words in buffers. One buffer is a working register which does all the syllable alignment. The other buffer is a latch which holds the second word until the code register has emptied. The code register is loaded such that shifting one bit position effectively shifts the code word 1-byte (syllable). By this method of shifting and by using the 50ns clock, the PC has the potential to process 4 syllables in one machine cycle.

The PC also utilizes shift registers to monitor the present program syllable index (PSI). By shifting a bit in a PSI register whenever the code register is shifted, the PC can keep track of the byte index into the code word. This counter enhances performance since no decoding of PSI value is necessary for control purposes.

Because of the shift method for extracting syllables, special circuitry is necessary for PSI initialization. To accomplish this, another counter is used, that is loaded with the decoded new PSI value. By forcing the code register and PSI ring counter to shift as the initial PSI ring counter decrements, the proper alignment is accomplished in the code register.

The PC must keep track of the PSI at the beginning of every operator for possible use by the SLC. To do this, the present PSI is stored in a register at the beginning syllable of every operator. The output of this register is the next operator PSI history. This information is available to the SLC and is also stored again in a different register when the next operator is started. The output of this register becomes the present operator PSI history and is also available to the SLC. The PC also checks the tag of the code word and generates an error condition when the SLC executes an incorrect code word. The PC also manages a register which indicates the Present Operator PWI value. This register is loaded by the SLC when a Branch Condition has taken place. The PC adds 1 to this register whenever an operator that just started has crossed a Word Boundary relative to the last operator executed.

The PC makes its own memory accesses. When the code register empties, the next word in the buffer is loaded in the code register and a PC memory request is sent to the Memory Control. Figure 5-9 is a functional block diagram of the Syllable Prep Module.

OPERATOR MAPPING MODULE

This module contains a 1K by 24 PROM, an output register, and parity checkers. The PROM is addressed by the selected syllable from the Syllable Prep module and also by two feedback bits from the output register. These two bits are the most significant bits of address and divide the memory into 4 256 address portions. The four sections of memory contain data for primary mode, edit mode, and variant mode operators. One 256 address portion is not used (addresses 000 – 255). This memory contains basically an entry vector and control bits for each operator. The entry vector is the starting address to the algorithm in the microprogram which implements the operator. Figure 5-10 shows the Operator Mapping Module block diagram.

The edit and variant bits provide information to the control module and address the PROM. Edit, variant, and primary mode operators are housed in the same memory. All locations in the variant and edit sections of the PROM memory always return to the primary mode.

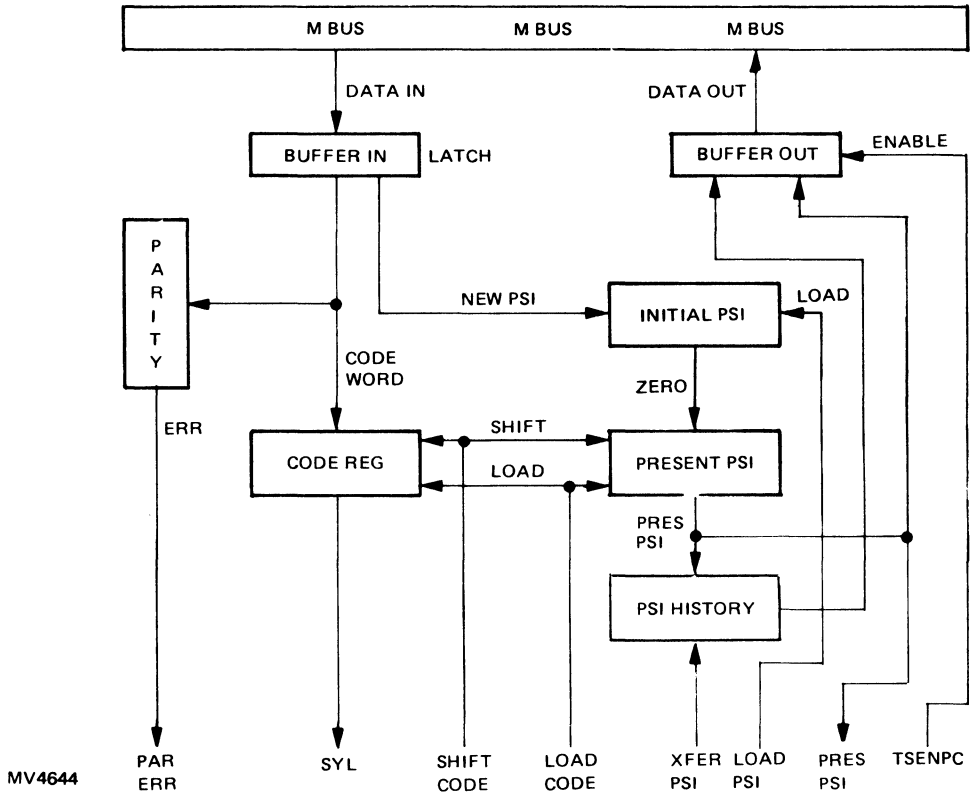


Figure 5-9. Syllable Prep Module Block Diagram

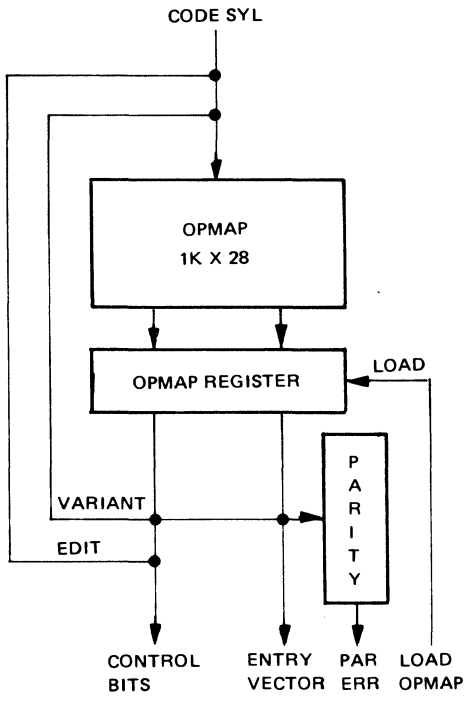


Figure 5-10. Operator Mapping Module Block Diagram

PARAMETER PREP MODULE

This module contains registers to build and hold parameter words. After a PROM output register is loaded with the location addressed by the operator code, the syllables of parameters, if any, are loaded one by one into the parameter building register. When the last parameter syllable is loaded, the PC is through with that operator and will have the code register pointing at the operator code of the next operator and waits for the EOP. The Parameter prep module is diagrammed in Figure 5-11.

After an EOP, the system clock starts the PC on the next operator. This also causes the contents of the building register to load into the holding register. The holding register provides the appropriate delay for the operator being executed to obtain its parameter word.

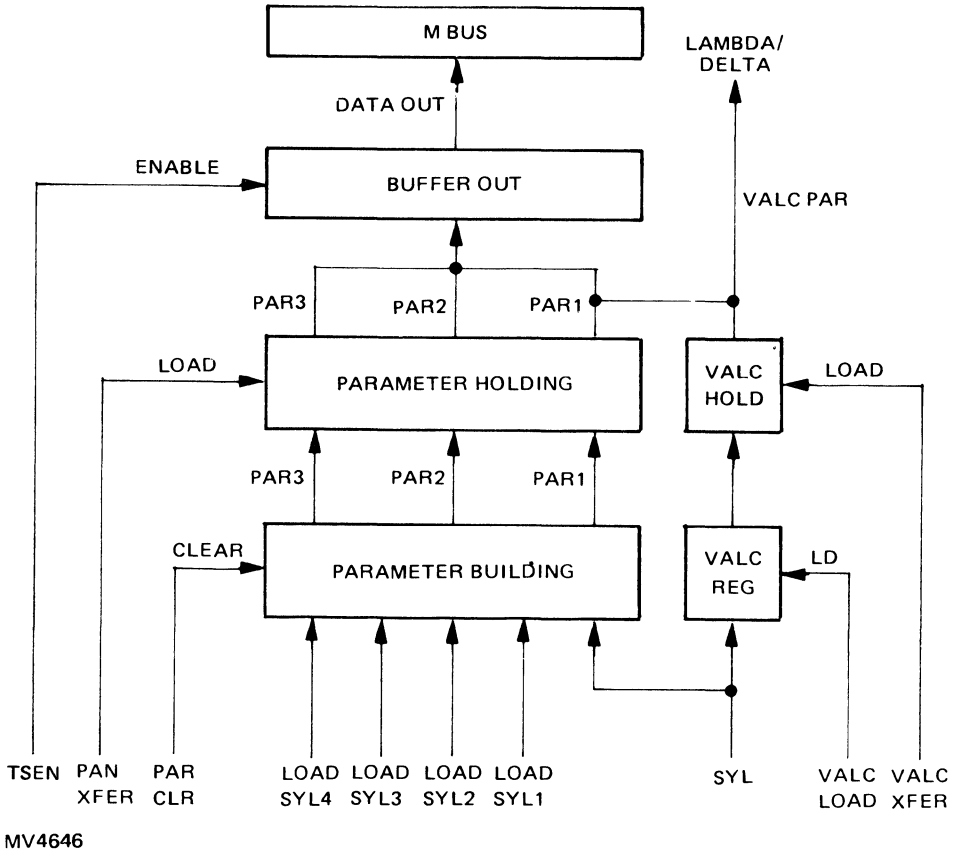


Figure 5-11. Parameter Prep Module Block Diagram

PC CONTROL MODULE

This module is responsible for the timing and control of the PC. Relying on the fast clock for performance, the PC must keep track of the relationship of the fast clock to the system clock. To do this, a ring counter is clocked with the fast clock. This provides phases of the system clock. The ring counter is synchronized to the system clock.

Another critical timing period is the machine cycle immediately following an EOP. This indicates the first clock cycle of the next operator to the PC and to commence decoding. The signal which accomplishes this is called SYNC/ and is set by a system clock after an EOP.

The PC must have a signal which indicates that the operator is finished being decoded and the parameter word has been built. The DONE/ signal does this and renders the other modules inactive until SYNC/ goes active again.

To prevent the Syllable Prep module from running ahead of its memory accesses after completing a code word, the PC waits until memory supplies another code word. The signal IDLE/ renders all modules inactive until memory responds with PCLD/. Then modules again are active until they are DONE/.

MEMORY AND MEMORY CONTROLLER

The B 5900 memory is a high speed, single bit error correcting RAM memory with storage capability of up to 512 thousand words (1 million if extended memory option is utilized).

Memory Control boards 1 and 2 are located in the B 5900 CPU backplane. Memory Control board 3 is located in the B 5900 memory backplane along with 5, 10, 15, or 20 RAM storage boards, a Regulator board, and a Terminator board.

MEMORY CONTROLLER FUNCTIONAL DESCRIPTION

The MC receives commands from the SLC by way of the 30-bit C-bus. In addition, the Program Controller may request a memory cycle by way of a direct interface. Memory Controller address, Write data, and Read data are transferred between the B 5900 system and the MC on the M-bus. Various control signals between the MC and other system modules are communicated by way of direct interfaces.

Memory Controller board number 1 (MC/1) contains Write and Read data latches and buffers as well as Read data error correction and detection circuitry. Write data latches store write information from the M-bus upon receipt of a Write Data Available signal on the C-bus. The latched Write data and the check bits are buffered and transferred on the M-bus. Read data syndrome bits are generated from the Read data and Read check bits to determine if an error is present. Single bit and multiple bit errors are detected and single bit errors are corrected, when error correction is enabled. The corrected Read data is transferred to the M-bus. Odd parity is checked on M-bus transfers to MC/1, except during maintenance modes, and is generated for M-bus transfers from MC/1.

Memory Controller board number 2 (MC/2) contains the control signal and address interface between the Memory and the B 5900 system. Cycle types are decoded and assigned priorities, and the cycle initialization is synchronized to the system clock. Control signals between MC/1 and MC/2 are transferred over two frontplane cables. Odd parity is checked and generated in the same manner as MC/1.

Memory Controller board number 3 (MC/3) generates the timing signals for all cycles (including Refresh) required by the storage boards, MC/1, and MC/2. A Cycle Start signal from MC/2, or a Refresh signal will enable the shift register that generates the internal timing for the Memory Controller.

The 128K X 12 RAM storage board assemblies contain 96 dynamic RAMs, and Read data, Write data, and control signal buffers.

The Regulator assembly contains a –5 Volt regulator, +12 Volt regulator and control sequence circuitry. Power on/off sequencing, over and under voltage detection/shutdown, current limiting and power interface signals are provided in this module. Power control signals interface to the Regulator by way of a backplane connector. Figure 5-12 represents the Memory Control module functional block diagram.

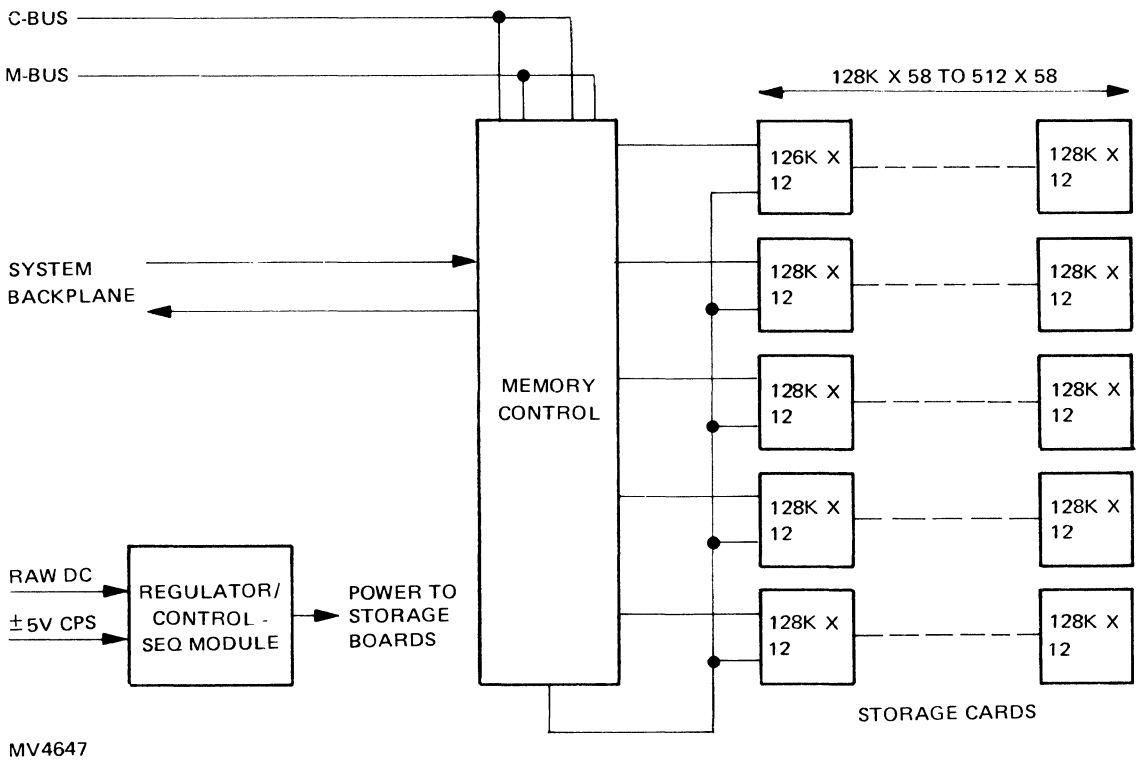


Figure 5-12. Memory Controller Block Diagram

Timing Section

The Timing Section accepts interface control signals from the B 5900 CPU and generates all timing signals for all cycles for the Module Control Assemblies, storage board assemblies, and the MC output interface. If the decoded operation control signals from the CPU indicate that a memory cycle is requested, and an abort present cycle signal is not received, the cycle type signals will cause the appropriate timing sequence to be initialized. The Timing Section also contains the 15 microsecond memory-refresh counter and gating signals.

MESSAGE LEVEL INTERFACE PORT (MLIP)

The MLIP functions are divided into the hardware module groups as shown in the MLIP block diagram, Figure 5-13. The modules are the following:

Controller
Status Tester
Longitudinal Parity Checker
Strobe Logic
Timer
MLI Bus Transceiver
Parity Generator/checker
MLI/MLIP Word Selector
Odd/Even Byte Selector
MLIP Write/Read Buffer
16-Bit Word Selector
C-bus Decode
C-bus Register

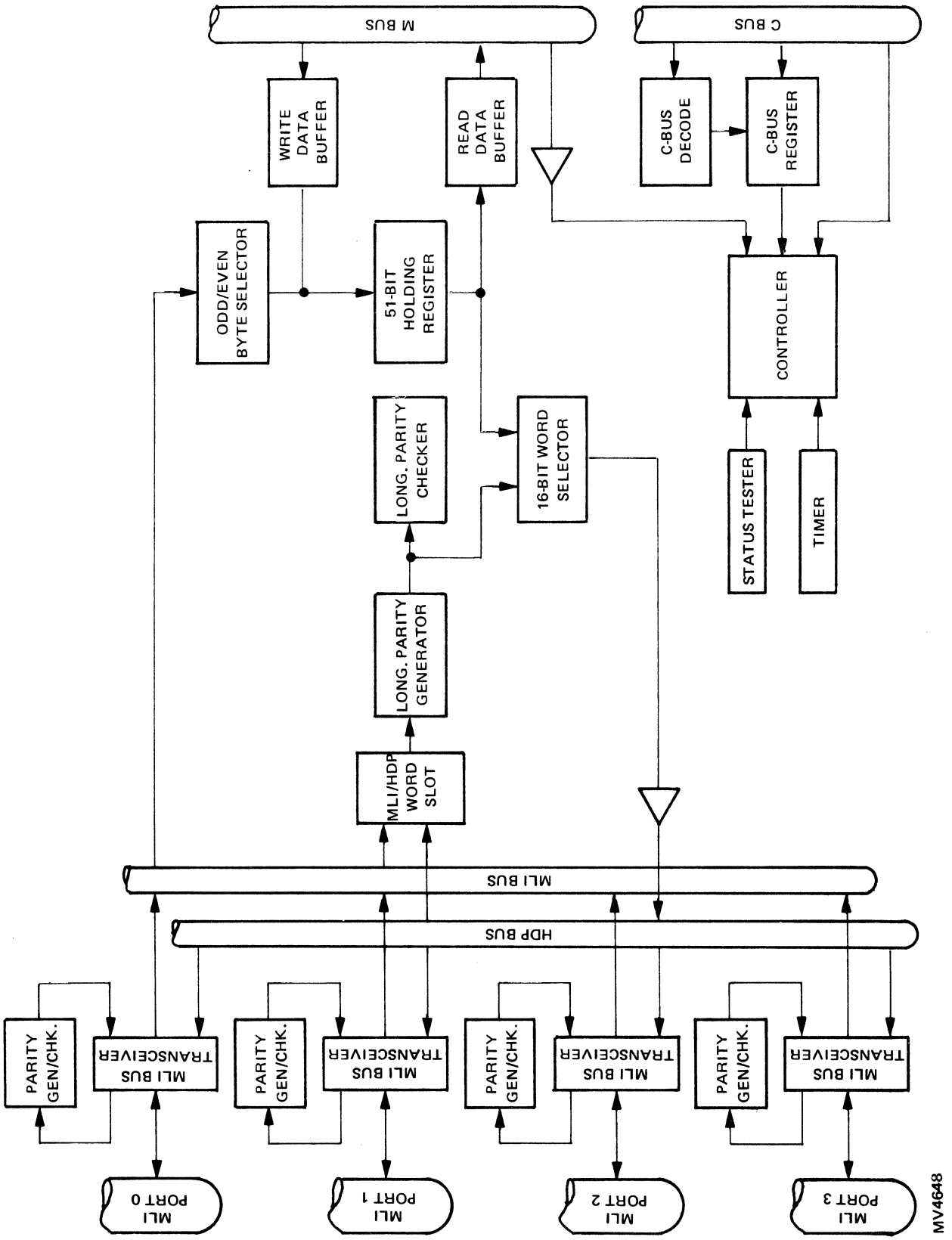


Figure 5-13. Block Diagram of the Message Level Interface Port

CONTROLLER

The Controller handles the asynchronous communications between the CPU and the MLI, the special timing and protocol required to send and receive information from the MLI and, on command from the SLC, the proper assembling and disassembling of the 16-bit word of information.

The SLC initiates the SEND cycle with two sequential commands. During the first command, the SLC presents 51 bits of information on the M-bus and a command to the Controller to initiate a SEND cycle. The SLC supplies on the M-Bus the port number, the start byte location, and the number (0-8) of bytes to send. The next command requires the SLC to supply on the M-Bus the data required to send to the MLI. The SLC can then continue its normal sequencing. The Controller stores the M-bus word in the Register and sends 16-bit words starting at the location specified for the number of bytes requested.

To initiate the READ cycle, the SLC sends a command to the Controller. The SLC supplies on the M-Bus the port number, the start byte location, and the number (0-8) of bytes to be read. The SLC can then continue normal sequencing.

The Controller reads the 16-bit MLI-bus into the MLI data Transceiver asynchronous to the MLIP timing. Upon the detection of the MLI data, the Controller will store the word into the bit field specified by the SLC and decrement the receive count. When the count reaches 0, the FINISH signal is generated. The SLC can then initiate another read command and simultaneously read the MLIP data.

Following the SEND and READ cycles, the SLC sends a command to the MLIP Controller to send the Longitudinal Parity Word (LPW). The SLC supplies on the M-Bus the port number the start byte location of 0, and two bytes to be read. The MLIP will send the value of the LPW to the MLI with the appropriate protocol.

STATUS TESTER

This module compares the status of the MLI previous state with that of the present state transmission. This module will make the comparison at the time of the Data Link Processor strobe. The previous state status is updated at the time of the Start I/O (SIO) strobe.

LONGITUDINAL PARITY GENERATOR

When commanded, this device will generate an accumulative parity for each vertical bit for the transmission and reception of MLI data. The Controller is in command of this module. A command from the SLC is used to clear the previous parity value, and a command from the MLIP Controller will generate the vertical parity of the input in conjunction with the parity of previous inputs.

LONGITUDINAL PARITY CHECKER

This device will test the LPW generated by the MLIP versus the one received from the DLP. The test is accomplished by generating the LPW of the incoming value with the stored value.

STROBE LOGIC

This device is designed to synchronize the DLP strobe with the system clock.

TIMERS

Two analog timers are used in the MLIP. The first timer is used to timeout the DLP response to a MLIP command. The timeout period is preset to 8 microseconds. The timer is set by the MLIP strobe IO signal and reset by General Clear. The second timer is used to timeout the DLP busy. The timeout period is preset

to 2 seconds. The timer is set by a SLC command and reset by General Clear. The timers generate a condition of a timeout error in their normal reset state.

MLI BUS TRANSCEIVER

This device connects outputs from the MLIP and inputs to the MLI bidirectional data bus. It is under command of the MLIP Controller.

PARITY GENERATOR/CHECKER

This module generates the odd parity to be sent with the MLI write data and tests for odd parity on the MLI read data.

MLI/MLIP WORD SELECTOR

This module determines the input for the longitudinal parity generator whether sending or receiving MLI data.

ODD/EVEN BYTE SELECTOR

This module determines if the start location for loading the system memory word is odd or even.

MLIP WRITE/READ BUFFERS

These modules transfer the M-bus data in and out of the MLIP register. The write buffer is controlled by the MLIP Controller while the read data buffer is controlled by the bus controller.

16-BIT WORD SELECTOR

This module, under command of the Controller, reads a 16-bit word from the 51-bit register and also the LPW. This determines the MLI word for transmission.

C-BUS DECODE

The module monitors the C-bus destination code. When the proper code for the MLIP is detected the decoder enables the C-bus register to read the C-bus commands.

C-BUS REGISTER

This module stores the commands for the MLIP from the control bus.

MAINTENANCE PROCESSOR AND INTERRUPT TIMERS

The following modules are shown in Figure 5-14.

The Interrupt Controller
The Error and Event Logic
The Maintenance Processor

The first two modules contains all external interrupt and microinterrupt logic.

There are 3 interrupt counters:

Interval Timer (12 bits)
Loop Timer (one shot approximately 2 seconds)
Time of Day (TOD) Timer (36 bits)

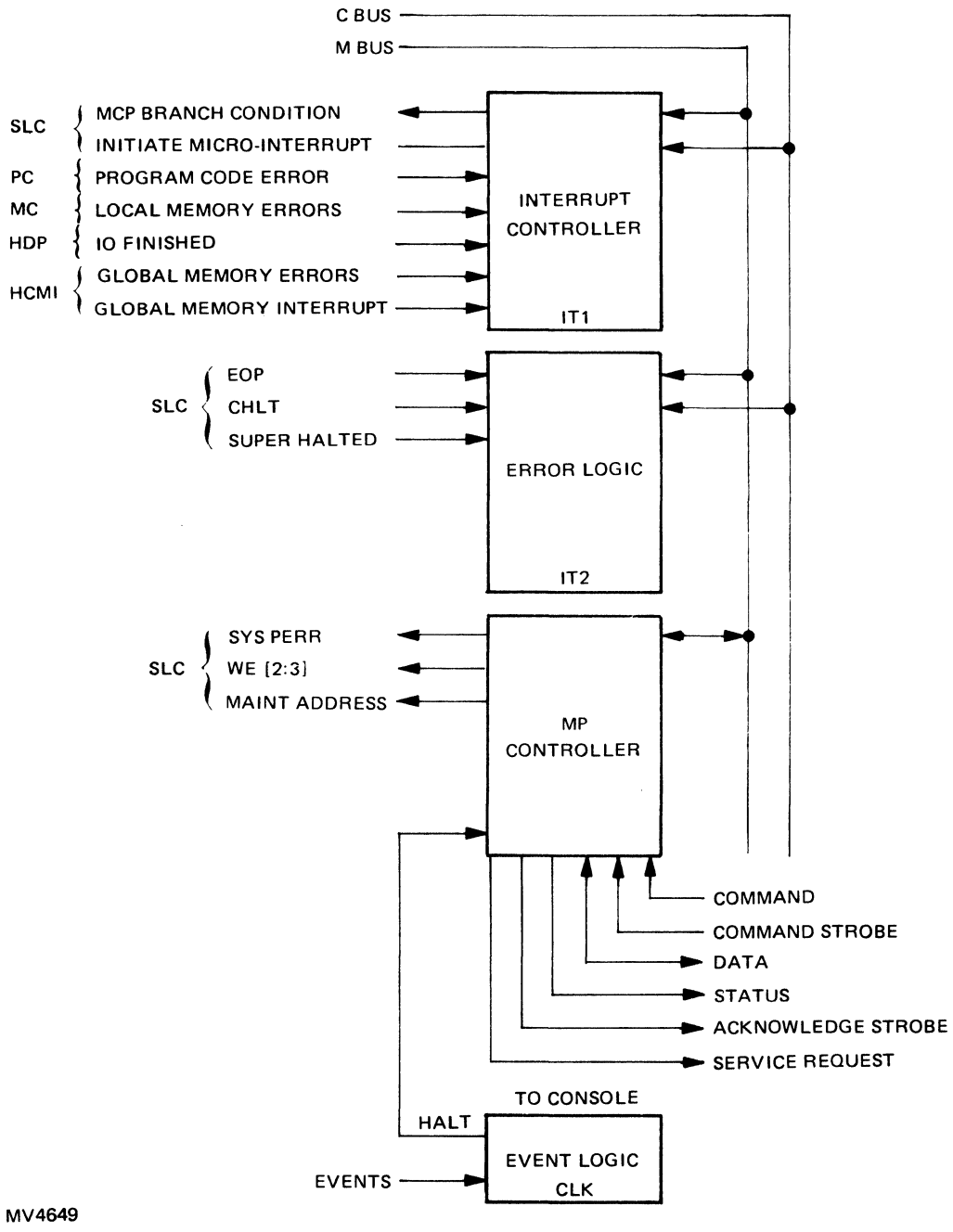


Figure 5-14. Maintenance Processor, Interrupt Controller, and Error Logic Modular Layout

The SLC sets and arms the Interval Timer. If the Interval Timer goes to 0 while armed, it generates an Interval Timer Interrupt and disarms itself. Whenever the SLC acknowledges an External interrupt, it automatically disarms the Interval Timer. Figure 5-15 shows the Interrupt Controller block diagram.

In normal mode (non-maintenance), the loop timer is a free running counter. Every occurrance of EOP sets it. If it times out, a Loop Timer interrupt is generated.

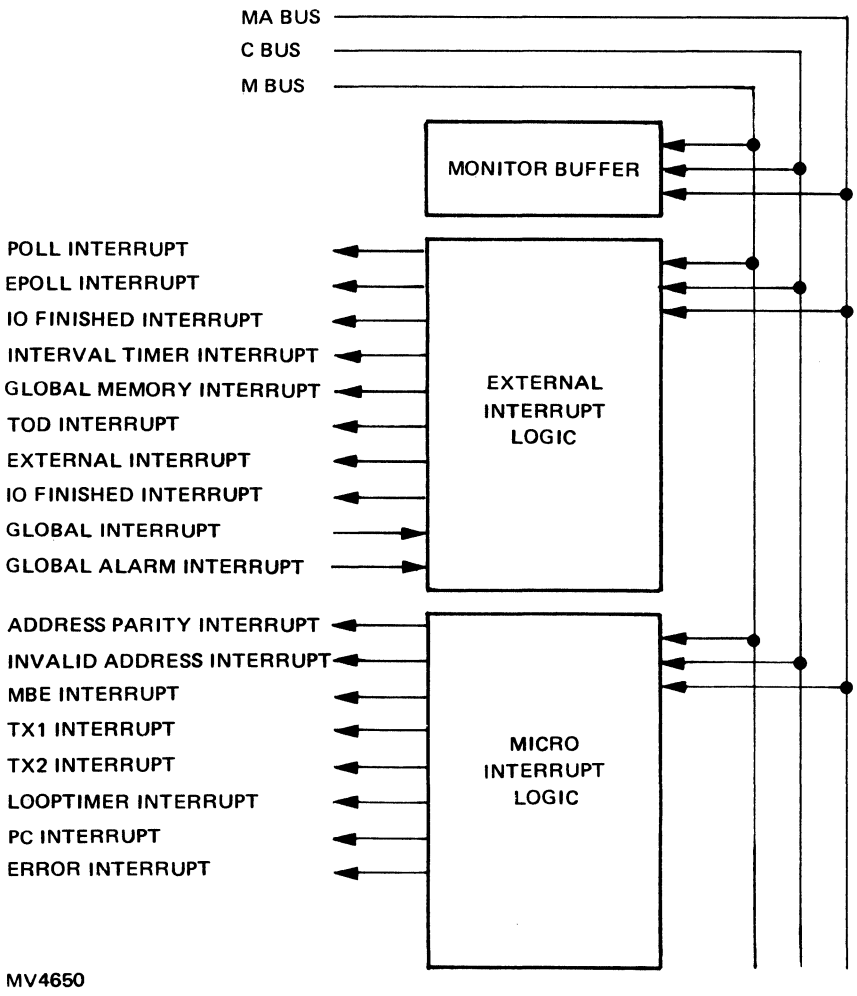


Figure 5-15. Interrupt Controller Block Diagram

The TOD counter is set by the operator from the Console. If TOD is not reset, it counts up to 45 hours. The External interrupts are sent to the SLC as branching conditions and they are handled in two groups. Within each group all conditions are sampled and cleared simultaneously.

Group 1 contains:

1. Interval Timer Interrupt (masked by Normal/Control State flip-flop)
2. IO finished Interrupt (masked by Normal/Control State flip-flop)
3. Global Memory Interrupt (masked by Normal/Control State flip-flop)

Group 2 contains:

1. POLLREQ Interrupt (cannot be masked by Normal/Control State flip-flop)
2. EPOLLREQ Interrupt (cannot be masked by Normal/Control State flip-flop)

The Normal/Control State flip-flop provides the common mask for all External interrupts. Loading of the Normal Control State flip-flop is done from the M-bus under control of the SLC. The microinterrupt logic collects all microinterrupt signals, and stores them in separate flip-flops as branching conditions to be used later by the SLC. The occurrance of a microinterrupt generates the signal "Initiate Microinterrupt" which forces the SLC to a fixed location that is common to all microinterrupt handlers.

The microinterrupts are as follows:

Address Parity Error
Multiple Bit Error
Invalid Address Error
Loop Timer
Program Error
Scan Bus Read Error (TX1)
Scan Bus Write Error (TX2)
Bus Detected Parity Errors (M-bus, C-bus, PROMs)

Only Global Memory Alarm can occur simultaneously with every one of the other microinterrupts. If two microinterrupts occur simultaneously, Global Memory Alarm has higher priority.

When any kind of interrupt routine is entered, a soft counter maintained in the register file is incremented by one. The counter is set to 0 when an EXIT or RETURN operator is executed, that is, a return from interrupt.

If the count reaches the value of five, the microprogram branches to the Superhalted state, and interrupts the console.

ERROR/EVENT LOGIC

This logic provides the means of halting the system clocks under Maintenance Processor control. Events are selected from the console and are active only if the Event logic is enabled. The event signals are described as follows and are primarily used for the maintenance function.

EOP
: This signal is generated by the SLC at the end of an operator. EOP will allow the system user to check the result of a single operator. This event is maskable by setting the correct mask bit in the clock snake.

Superhalted
: The Superhalted signal inhibits the system clock under MP control and is setable from the SLC. Superhalt is non maskable.

HOLD-1/
: This signal is valid at the end of a memory cycle. HOLD-1/ is maskable from the clock snake.

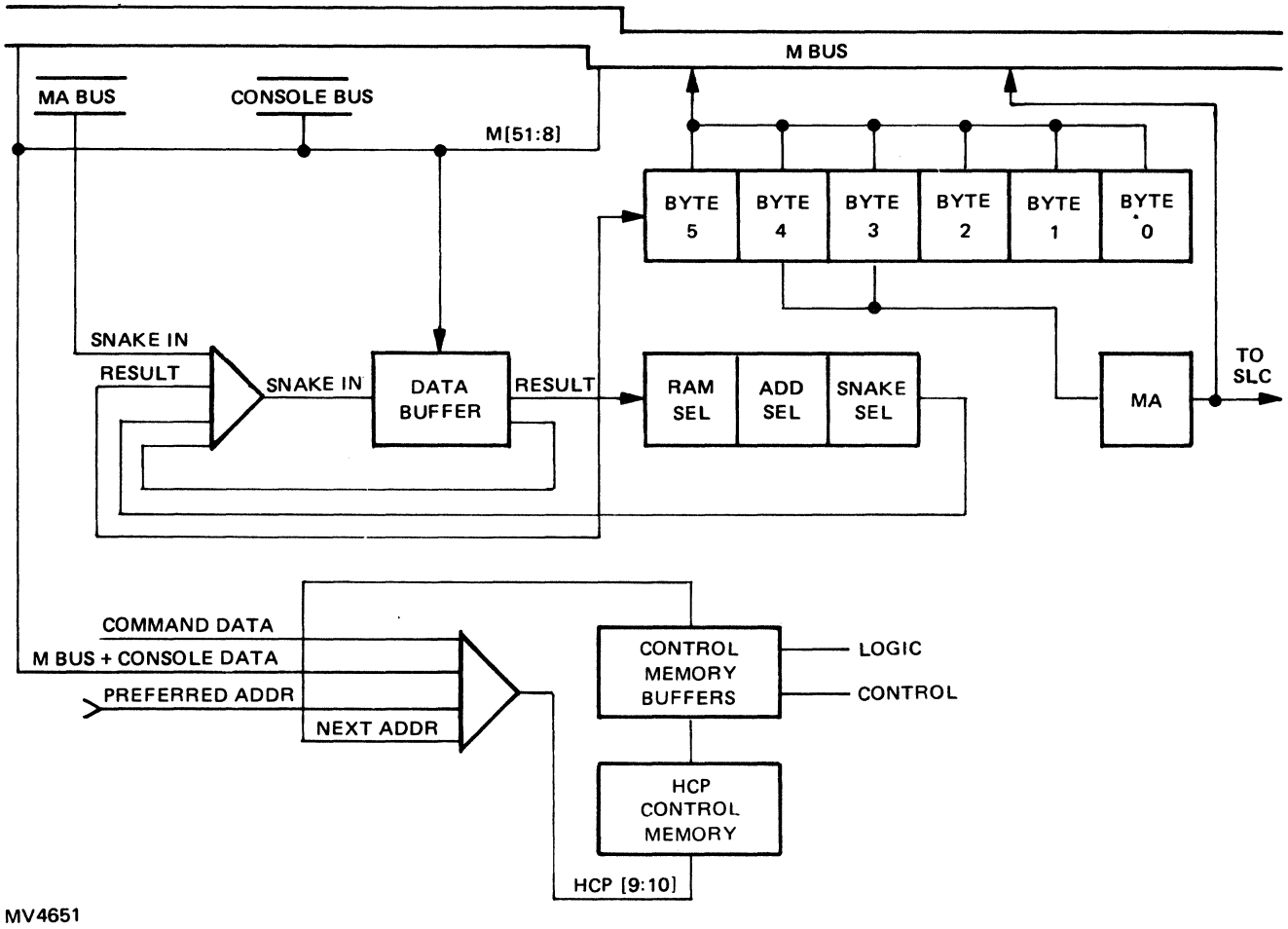


Figure 5-16. Maintenance Processor Control Logic

MAINTENANCE PROCESSOR CONTROL MEMORY

The Control Memory contains command vectors necessary to execute the MP Control algorithms. A command vector consists of a command field and a next address field. The command field contains the control signals needed on the various internal and external interfaces for the proper execution of the commands. The next address field, together with branching conditions generated internally and in other modules, determines the address of the next command vector.

DATA EXCHANGE LOGIC

The Data Exchange Logic consists of an 8-bit data buffer that can be inserted serially between the input and the output of any shift register of the system that is under the control of the Maintenance Bus.

The Data Buffer is also connected in serial fashion to a 44-bit shift register called the HCPBUFFER. The HCPBUFFER provides data to and reads data from the M-bus as well as the maintenance address register that is under the control of the MP Control Memory.

The Data Buffer also transmits and receives data and control to and from the Console. The combination of the MP Control Memory and the Data Exchange logic allows the Console to read ad set any storage element of the system, and thus makes the following operations possible:

AlterDisplay any register or memory word
Initialize the system to any given state
Run System Test from the Console

SECTION 6
B 5900 INPUT/OUTPUT DATA COMMUNICATION SUBSYSTEM

MLIP GENERAL INFORMATION

The MLIP is a semi-autonomous control device, which is used to create and control interfaces between the software Master Control Program (MCP) and the Input/Output Data Communication (IODC) subsystem. Semi-autonomous means that the MLIP must be initiated into operation by the MCP, through execution of a CUIO operator code. Once the MLIP is initialized into operation, the microprogram takes command, and subsequent MLIP operations are determined by the microprogram logic, rather than by the MCP.

In addition to creating and controlling IODC interfaces, the MLIP also performs other system functions that involve the use of timers or time-counting circuits. Some of these timing functions are controlled by inputs to the MLIP from the software operating system and others are automatic functions of the MLIP logic circuits.

IODC SUBSYSTEM GENERAL INFORMATION

Peripheral I/O devices in a B 5900 system (see Figures 6-1 and 6-2) are controlled by Data Link Processor (DLP) adapters. A unique DLP adapter is used for each type of peripheral device connected to a B 5900 system. A DLP adapter contains micro-coded control programs which are unique to the type of I/O device the DLP controls.

DLP adapters are card-package modules which plug into a IODC-Base module backplane of the Central Processing Cabinet. The IODC subsystem is connected to the MLIP module of the CPU by means of external 25-wire Message Level Interface cables. There are from 1 to 4 MLI cables connected to the MLIP module, and each MLI cable interfaces one IODC-Base module to the MLIP module. (See Figure 6-3.) The system is able to run on just one UIO base module. Two are used to preclude inherent delay due to required intercommunication between an NSP and an LSP that would have to be located in the same base module. Also, this would require an additional distribution card and path select module.

The 25-line MLI cable contains 17 lines used to transfer a word of data (16-bits of data plus an odd-parity bit) between the MLIP and the UIO subsystem. This cable also contains 4 lines used to send DLP sequence counts and result status to the MLIP module. One line of the MLI interface is a system strobe-signal line, used by the MLIP to initiate actions in the IODC subsystem logic. Another line is a strobe signal line used by the IODC subsystem to to initiate actions in the MLIP logic. The remaining 2 lines are used for various synchronizing logic levels and signals, during an MLI line communication process.

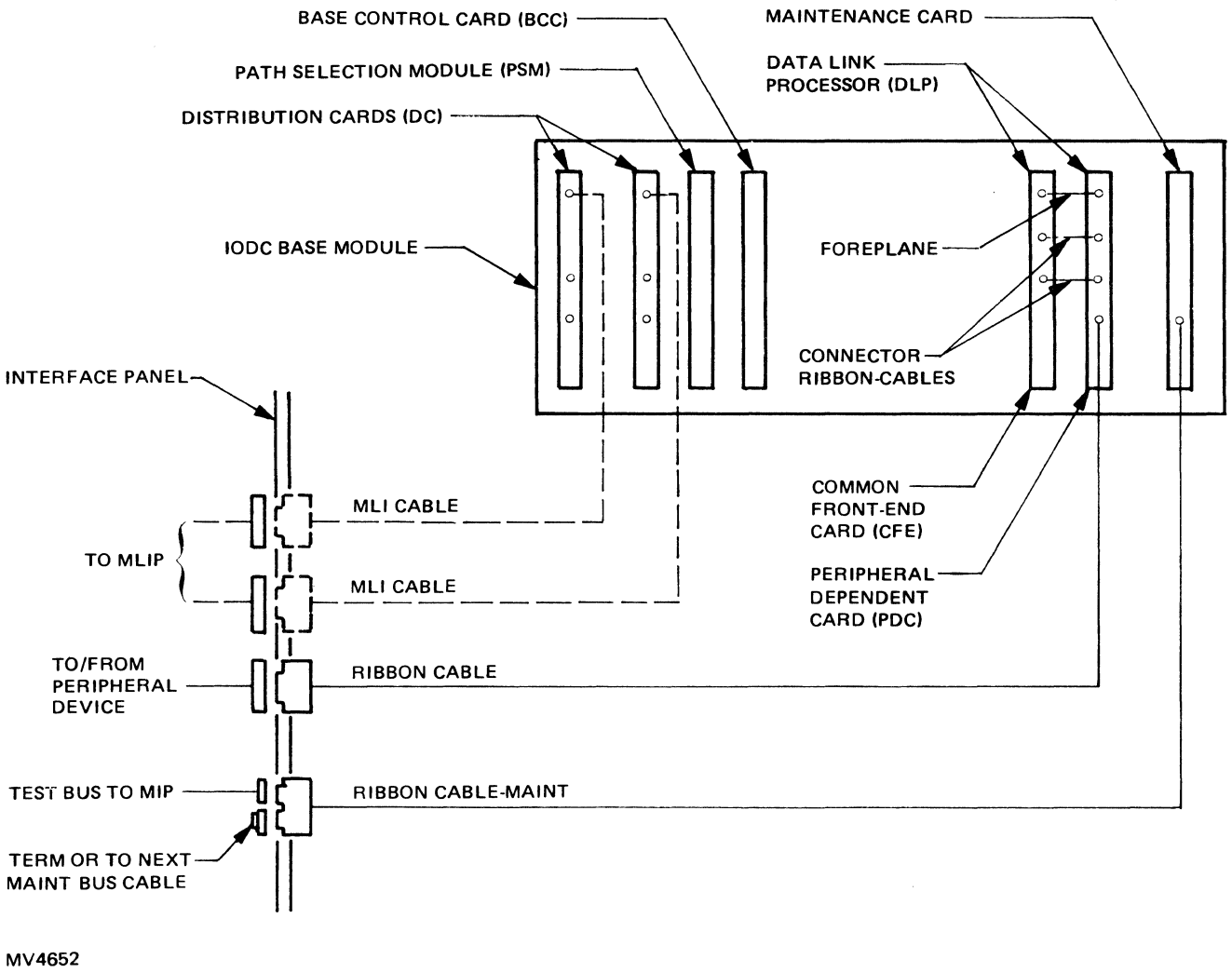


Figure 6-1. IODC Base Module With One DLP

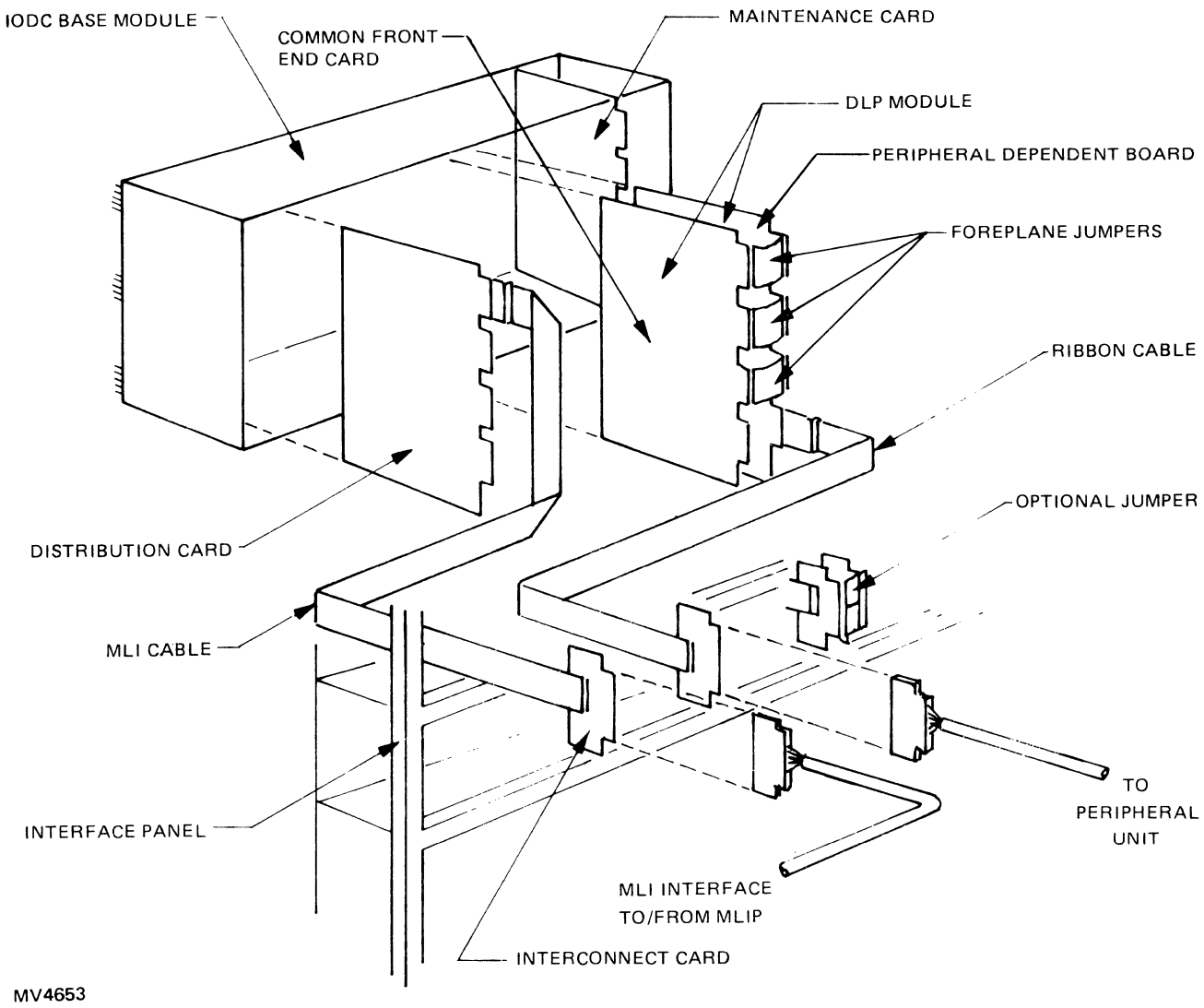
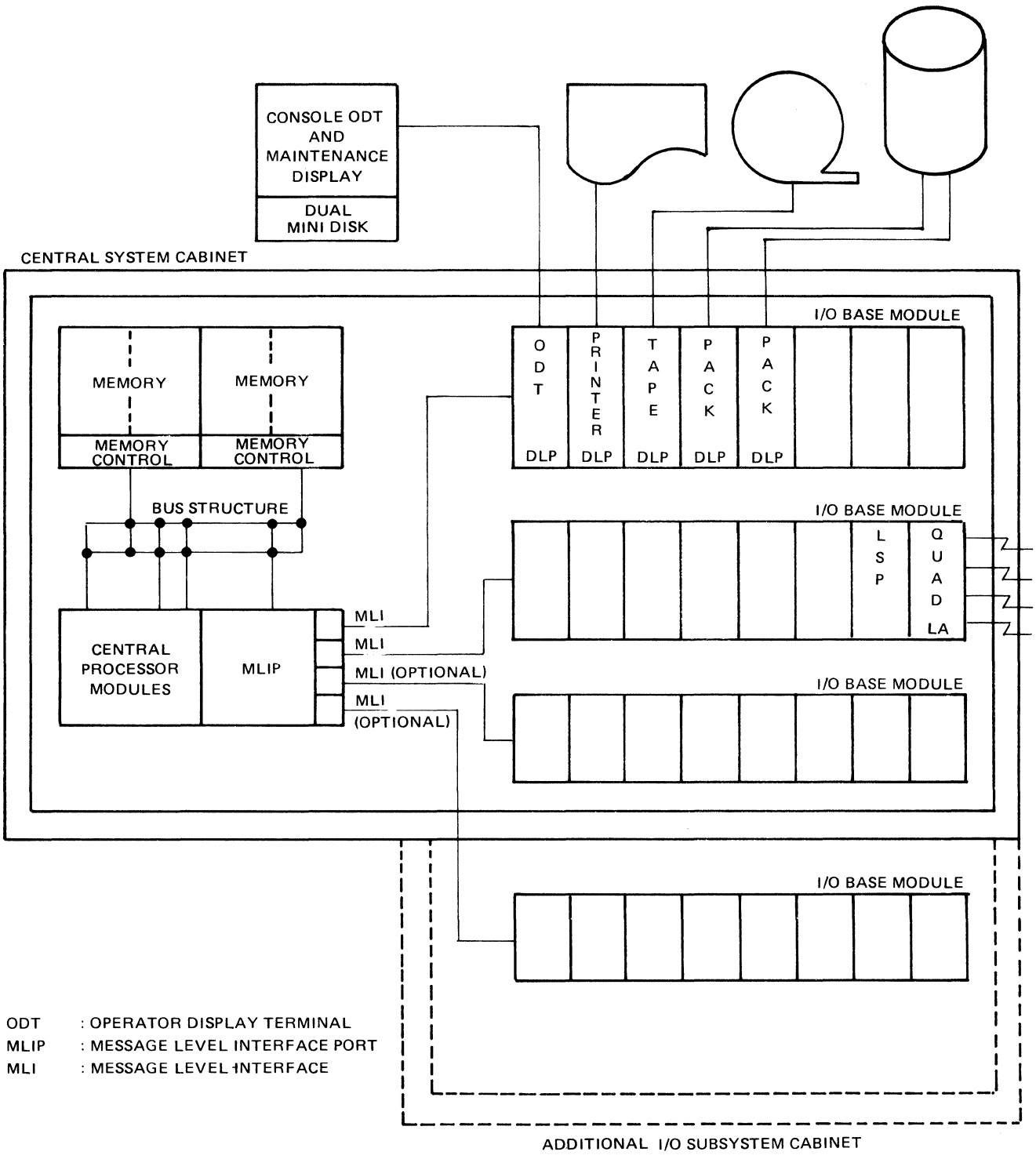


Figure 6-2. B 5900 IODC Base Module Organization



MV4613

Figure 6-3. B 5900 IODC and System Organization

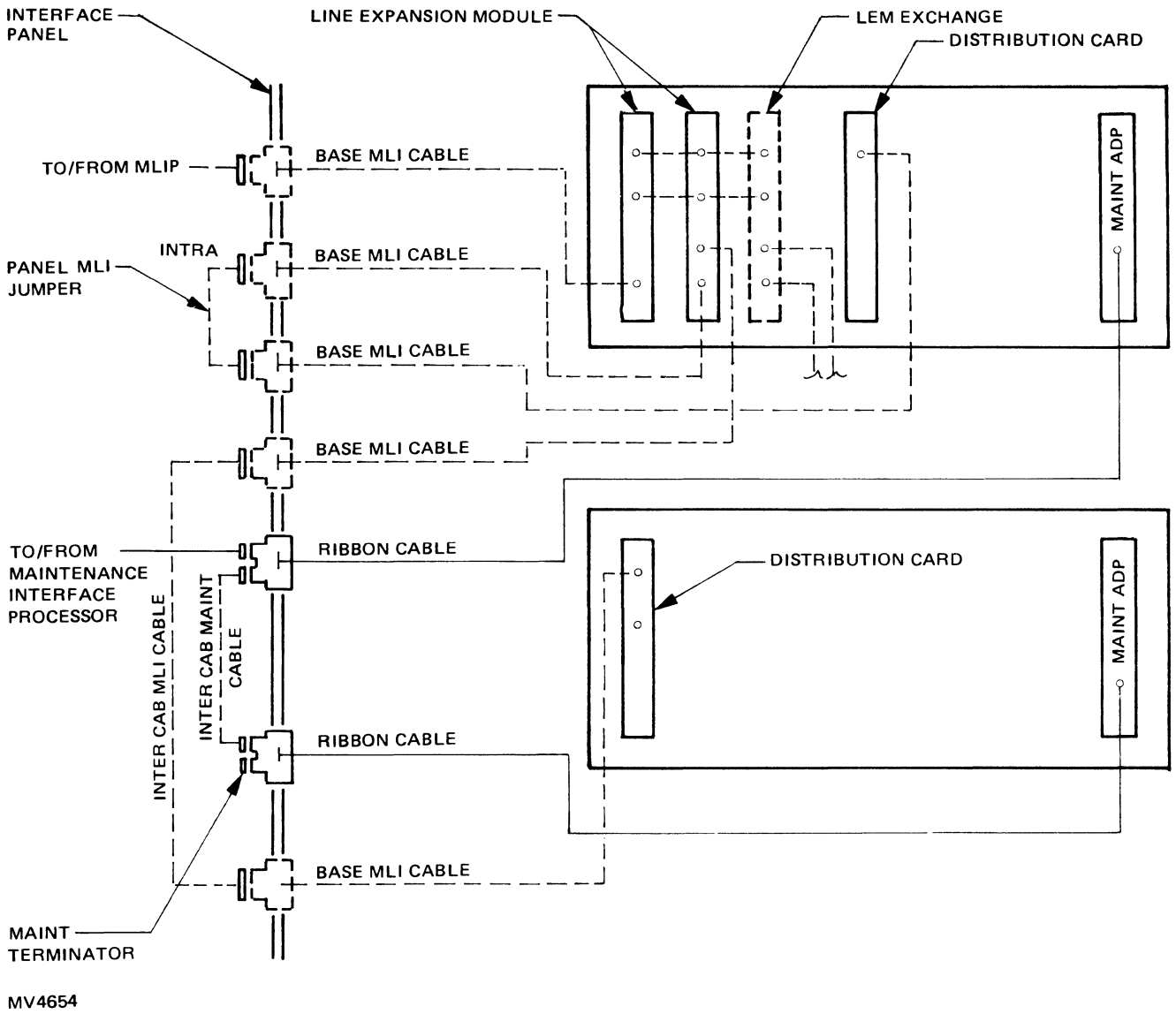


Figure 6-4. Multiple IODC Cable Connections

B 5900 I/O DEVICE OPERATION PROCESSES

I/O peripheral device operations begin when the MLIP module uses one of its MLI interface cables to communicate with the IODC subsystem. This interface communication must follow an established MLI interface protocol. The protocol requires that at least 4 MLI data words of IODC control information, in fixed word formats, be passed from the MLIP module to the IODC-Base and DLP logic. The entire process of meeting the requirements of the MLI protocol is commonly referred to as a connection sequence.

The 4 required data-words transferred during a connection sequence are:

An MLI Address word, which identifies the Base module and DLP addressed.

Two Descriptor-link words, which identify the MLI (host system) making the connection, and contain the memory address of the Input/Output Control Block that initiated the connection sequence.

A Longitudinal Parity Word, which is used to verify that the preceding 3 words are valid connection sequence words.

An MLIP module I/O device initiation process begins when the Data Processor module executes a CUIO operator code, and transfers the starting memory address of an IOCB to the MLIP logic. The MLIP then causes a connection sequence to be performed.

I/O OPERATIONS

Input/output operations are provided by the Message Level Interface Port (MLIP). The MLIP interfaces with the software through the CUIO operator and the I/O Finished external interrupt. This section describes the details of that interface. The MLIP also communicates with the Message Level Interface.

The CUIO operator is used to inform the MLIP about one or more I/O operations (IOCBs) which are to be performed.

A CUIO operator causes the CPU to evaluate a present, unpaged, unindexed Word DD on top of the stack and to verify that the referenced operand contains an IOCB mark. If the operand does not contain an IOCB mark, an invalid operand interrupt will be generated. If the operand contains the IOCB mark, the address of the operand will be passed to the MLIP. When the MLIP indicates that it has received the address, the CUIO operator will execute to completion.

The MLIP reads the memory word addressed by the CUIO operator and also verifies that the memory word contains an IOCB mark. (If the memory word does not, the MLIP will use the Error IOCB to indicate the error.) The MLIP verifies the validity of the Descriptor in the Command Queue Pointer word of the IOCB and also that the area it references is a Command Queue (by checking the control word for its queue mark) and then enqueues the IOCB (or the chain of IOCBs linked together) in the Command Queue. After the IOCB is enqueued, the MLIP determines if it should initiate the first command in the queue.

INTERRUPT

The MLIP raises an interrupt line to indicate to the CPU that an I/O Result Queue requires attention. (This interrupt interface does not contain any reference to a specific IOCB or Result Queue.)

The MLIP will raise the interrupt line upon completion of an IOCB with any of following conditions:

- The Cause I/O Finish Event bit of the MLIP control field in the IOCB is set.
- The Exception bit of the MLIP result in the IOCB is on. The CPU is executing the IDLE operator (optional).
- The Command Queue of the completing IOCB is empty.

SHARED DATA STRUCTURES

All communication between the MCP and the MLIP is through the use of the defined structures called I/O Control Blocks (IOCBs) and Command Queue Headers. An IOCB contains the information necessary to perform one I/O operation. A Command Queue Header contains the information necessary for the MLIP to maintain the queue structures. Each IOCB can be linked with other IOCBs to form two types of linked structures: Command Queues and Result Queues. The queues allow the MLIP to execute chains of separate I/O operations without direct MCP intervention. Command Queues are referenced by way of Command Queue Headers. Result Queues are referenced by way of Result Queue Heads.

Figure 6-5 depicts a structure with five IOCBs. IOCBs #1 and #2 represent I/Os which have completed and are linked together in one Result Queue. The Result Queue is Last In First Out (LIFO); in this example, IOCB #1 completed before IOCB #2. IOCB #3 is active and has been removed from the Command Queue. IOCBs #4 and #5 are linked in the Command Queue. IOCB #4 will be initiated next.

In addition, Command Queue Headers can be linked together in a Horizontal Queue.

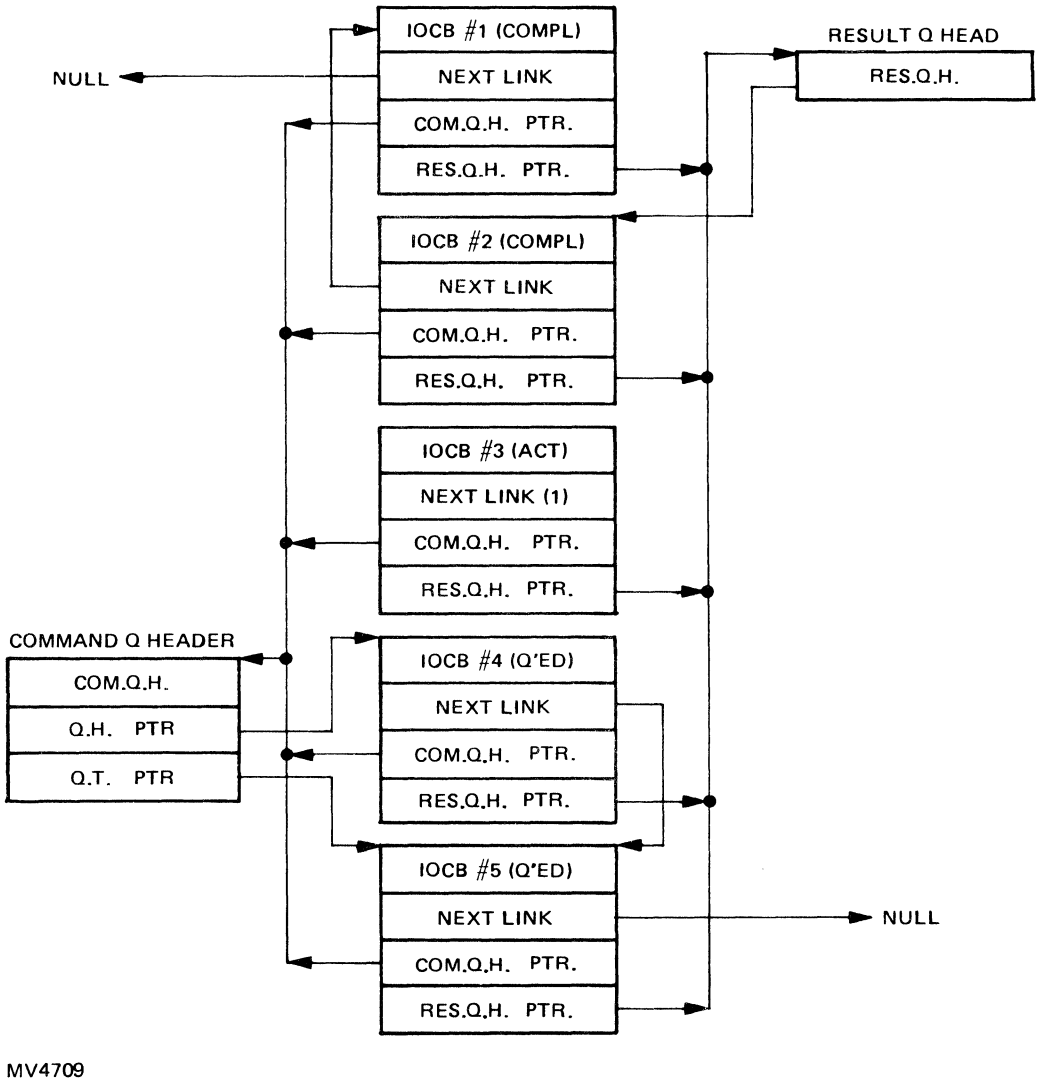


Figure 6-5. Command and Result Chaining

IOCB

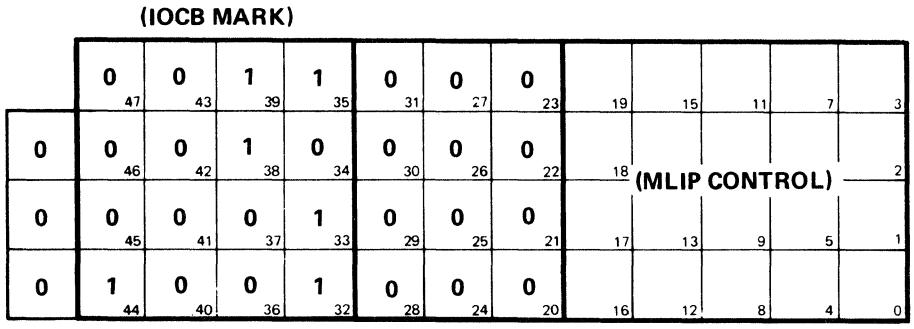
The IOCB is a combination of MCP and MLIP work areas that contain both Data Descriptors and formatted operands. The Data Descriptors must be present and unpaged. The format of the IOCB is shown in Figure 6-6.

0	[CW] :	CONTROL WORD
1	[DLPAW] :	DLP ADDRESS WORD
2	[CCHP] :	COMMAND QUEUE HEADER POINTER
3	[SELFP] :	IOCB SELF POINTER
4	[DLPCP] :	DLP I/O COMMAND POINTER
5	[DLPRP] :	DLP I/O RESULT POINTER
6	[DLPCRL] :	DLP COMMAND/RESULT LENGTHS
7	[RM] :	RESULT MASK
8	[RQHP] :	RESULT QUEUE HEAD POINTER
9	[NL] :	NEXT IOCB LINK
A	[CDP] :	MLIP CURRENT DATA AREA POINTER
B	[CL] :	MLIP CURRENT I/O LENGTH
C	[MRSLT] :	MLIP STATE AND RESULT
D	[STIME] :	I/O START TIME
E	[FTIME] :	I/O FINISH TIME
F	:	MCP
	• • •	

MV4710

Figure 6-6. IOCB Format

Word zero, the Control Word, is a formatted operand containing the sixteen bit "IOCB mark" and the MLIP control field. It is initialized by the MCP and remains unchanged during the I/0. Figure 6-7 depicts the IOCB Mark hex 10CB.



MV4711

Figure 6-7. IOCB Mark – 10CB

MLIP Control Word

Format of the MLIP Control field is given below.

[0:1] Queue at Head
If this bit is on during the execution of the CUIO operator, the MLIP will insert the IOCB (or the chain of IOCBs) at the front of the Command Queue.

[1:1] MLIP/DLP Command
If this bit is on, the IOCB contains a command to be interpreted by the MLIP. When this bit is off, the command is directed to the DLP identified by the DLP Address Word. The exact nature of the command is indicated in the DLP I/O Command, which is accessed by way of the DLP I/O Command Pointer.

[2:1] Attention
If this bit is on, at the completion of the operation the MLIP will build an MLIP result that has both the attention and exception bits on.

[3:1] Cause I/O Finish Interrupt
If this bit is on, the MLIP will cause the I/O finish CPU interrupt at the completion of the operation. If this bit is off, the MLIP will decide to cause the I/O finish interrupt using other criteria.

[4:1] Memory Override/Memory protect
If this bit is on, the MLIP will ignore the tag of memory words during transfer. If this bit is off, the MLIP will stop data transfer if attempting to read or write an odd-tagged word.

[5:1] Input
If the DLP goes to Read status and this bit is not on, the MLIP will flag a DLP status error and not allow the data transfer.

[6:1] Output
If the DLP goes to Write status and this bit is not on, the MLIP will flag a DLP status error. To effect the echo command, both the Input and Output bits must be on; if both bits are off, an attempt to transfer data will cause a DLP status error and no data will be transferred.

[7:1] Output Zeros
If this bit is on and the Output bit is on, the MLIP will send all zeros to the DLP. The MLIP will send bytes of zeros until its Current I/O Length counter is 0. It is an error if this bit is on at the same time the Input bit is on. The only valid incidence of "Output Zeros" is for magnetic tape erase. (Magnetic tape erase is the only operation where the DLP throws away the data that it gets from the host.) This feature is implemented as a general command so that the MLIP can remain transparent to the characteristics of the different DLPs.

[10:3] Tag Control
The values of this field define how the tag is handled during data transfer.

010 Transfer double byte tag.
Tags are treated as two additional bytes of data. On output, the 3 bits of tag are placed in the 3 least significant bits of the most significant byte; the other 13 bits are set to 0. On input, the 3 least significant bits of the most significant byte are placed in the tag bits of the word; the other 13 bits are ignored.

100 Force tags to single (0).
Tags are not treated as data to be transferred. On input, the tag of each word is set to 0. On output, the tag of each word is skipped.

110 Force tags to double (2).
Valid on input only, performs as force tags to single except the tag of each word is set to double (2).

111 Force tags to code (3).
Valid on input only; the tag of each word is set to code (3).

[11:1] Word/Character-oriented Transfer
If this bit is on, the MLIP Current Data Area pointer must be a word DD and the MLIP Current I/O Length word will describe the length of transfer in words. If this bit is off, the MLIP Current Data Area pointer must be an EBCDIC string DD and the MLIP Current I/O Length word will describe the length of transfer in characters.

[12:1] Memory Direction
If this bit is on, the MLIP will transfer data backward into memory. If this bit is off, the data transfer will be forward. If this bit is on and either the Word-Oriented Transfer bit or the Output bit is on, an invalid MLIP control field error will be returned.

[13:1] Continue Count at End of Length
When this bit is on (and the Input bit is true), the MLIP will not terminate data transfer when the MLIP Current I/O Length is zero; rather, it will allow the DLP to send data but will not store the additional data in memory. The Current I/O Length will continue to be decremented. If this bit is on and the Output bit is on, an invalid MLIP control field error will be returned.

[14:1] Ignore Count Error
If this bit is on, the MLIP will not set the Count Error Exception bit in the MLIP result field when the MLIP Current I/O Length is not equal to zero at the finish of the I/O.

[15:1] Don't Count
If this bit is on, the MLIP does not increment the active count when initiating the command or decrement the active count when it completes. This bit is intended for "cancel" operations.

[16:1] Ignore Suspend All Queues
If this bit is on, the MLIP will not suspend the IOCBs Command Queue when adding an IOCB to a Result Queue, regardless of the setting of the Suspend All Queues flag.

[17:1] Immediate
If this bit is on when the MLIP is looking at the first command in the queue (which occurs after the enqueueing step of the CUIO operator, because of an I/O finish, or while handling horizontally queued Command Queues), the MLIP will attempt to initiate the command regardless of the value of the active count or state of the Suspended bit of the queue. If the DLP is busy when the MLIP attempts to connect, the MLIP will, if necessary, attempt to add the Command Queue to a Horizontal Queue.

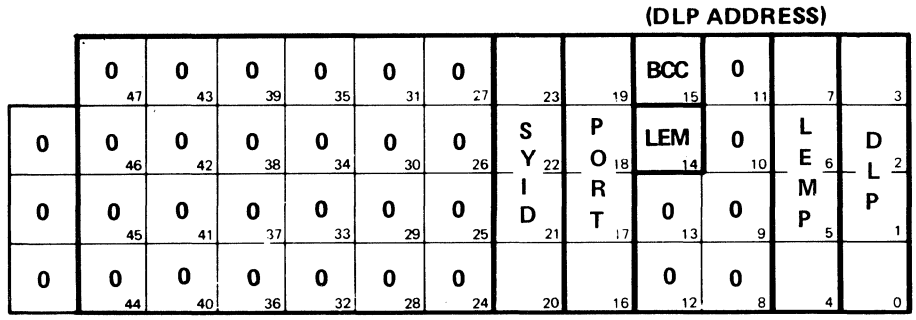
[19:2] Reserved
This field must be zero.

The following table gives all valid combinations of the MLIP control field. Generic classifications such as MLIP OP, TEST, do not necessarily correspond to such classifications as TEST and READ, of the first four bits of the DLP I/O Descriptor. The bit value X means that the bit can be either on or off.

Table 6-1. MLIP Control Field – Valid Commands

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	IMMEDIATE	IGNORE SUSPEND ALL QUEUES	DONT COUNT	IGNORE COUNT ERROR	CONTINUE COUNT	DIRECTION	WORD ORIENTED	TAG CONTROL			OUTPUT ZEROS	OUTPUT	INPUT	MEMORY OVERRIDE	CAUSE I/O FINISH	ATTENTION	MLIP/DLP	QUEUE AT HEAD
MLIP OP	X	X	X	0	0	0	X	1	0	0	0	0	0	0	X	X	1	X
TEST	X	X	X	0	0	0	X	1	0	0	0	0	0	0	X	X	0	X
INPUT	X	X	0	X	X	0	1	0	1	0	0	0	1	1	X	X	0	X
	X	X	0	X	X	0	X	1	0	0	0	0	1	X	X	X	0	X
	X	X	0	X	X	0	X	1	1	0	0	0	1	X	X	X	0	X
	X	X	0	X	X	0	X	1	1	1	0	0	1	1	X	X	0	X
	X	X	0	X	X	1	0	1	0	0	0	0	1	X	X	X	0	X
OUTPUT	X	X	0	X	0	0	1	0	1	0	0	1	0	1	X	X	0	X
	X	X	0	X	0	0	X	1	0	0	0	1	0	X	X	X	0	X
	X	X	0	X	0	0	X	1	0	0	1	1	0	X	X	X	0	X
ECHO	X	X	0	X	0	0	X	1	0	0	0	1	1	X	X	X	0	X

Word one, the DLP Address Word, is a formatted operand containing the address that the MLIP must send during the connection sequence to identify the path to the proper DLP. The conversion from unit number to address, the path to the unit, is done by the MCP when initializing the IOCB. Figure 6-8 shows the DLP address word.



MV4712

Port
Contains a binary value from 0 to 7 which identifies the MLIP port to be used.

LEM Port
Contains a binary value from 0 to 7 which identifies a UIO base connected to the MLIP port.

BCC
Indicates that the command is for the Base Control Card within the addressed UIO base.

LEM
Indicates that the command is for the Line Expansion Module.

DLP
In the absence of either the BCC or LEM bit being equal to one (1), contains a binary value from 0 to 7 which identifies the DLP within the addressed UIO base.

Note that if the DLP address specifies a non-present port, a "DLP NOT PRESENT" result will be returned.

Figure 6-8. DLP Address Word Format

Word two, the Command Queue Header Pointer, contains a present, unpaged, unindexed Word DD which points to the Command Queue Header. This word is initialized by the MCP and is unchanged during the I/O.

Word three, the IOCB Self-pointer, is a present, unpaged, unindexed Word DD which points to the IOCB, itself. This word is used by the MLIP whenever it has to link the IOCB into a queue. The IOCB Self Pointer is initialized by the MCP and remains unchanged during the I/O.

Word four, the DLP I/O Command Pointer, contains a present, unpaged Word DD which points to the area containing the DLP I/O Descriptor. The Descriptor may be either indexed or unindexed, depending upon the memory management requirements of the system. The DLP I/O Command Pointer and the data to which it points are initialized by the MCP and remain unchanged during the I/O.

Word five, the DLP I/O Result Pointer, contains a present, unpaged Word DD which points to the area into which the MLIP will place the DLP I/O result Descriptor. The Descriptor may be either indexed or unindexed, depending upon the memory management requirements of the system. The DLP I/O Result Pointer is initialized by the MCP and remains unchanged during the I/O.

Word six, the DLP Command/Result Lengths, is a formatted operand containing the length of the DLP command (sixteen bits) and the DLP result (sixteen bits) in bytes. The rest of the word is reserved. Both values must be an even number of bytes. The lengths are to be used by the MLIP as maxima when sending the command and receiving the result. The command length is DLP dependent and will have to be known by the MCP when initializing the IOCB. If the DLP continues to indicate "send Descriptor status" after the entire array has been sent, the MLIP will disconnect the DLP (leaving the DLP hung) and indicate an "unexpected DLP status" error to the MCP. The result length, which is DLP and operation dependent, will have to be known by the MCP when initializing the IOCB. If the result returned by the DLP is longer than the length provided by the MCP, the MLIP will disconnect the DLP (leaving it hung) and indicate an "unexpected DLP status" error. This word is initialized by the MCP and remains unchanged during the I/O. Figure 6-9 depicts the DLP Command/Result Lengths word format.

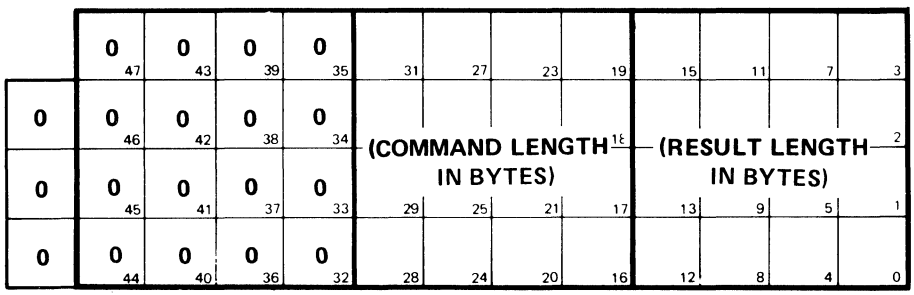


Figure 6-9. DLP Command/Result Lengths Word Format

Word seven, the Result Mask, contains a formatted operand. The DLP I/O result (first 48 bits) is ANDed with the logical complement of this word; if the result is non-zero, the Exception and DLP error bits are set in the MLIP result word. Note that the ANDing process does not affect the DLP result, which is returned to the host in the original form of the DLP result. The Result Mask is initialized by the MCP and is unchanged during the I/O.

Word eight, the Result Queue Head Pointer, contains a present, unpaged indexed Word DD which points to the Result Queue Head in the Result Queue Array. This word is initialized by the MCP and is unchanged during the I/O.

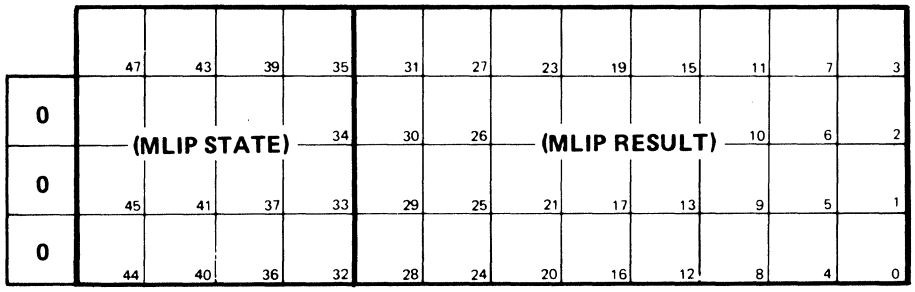
Word nine, the Next IOCB Link, contains either a present, unpaged, unindexed Word DD pointing to the next IOCB in the command or result chain, an operand with a value of 1 indicating that the IOCB has been initiated by the MLIP, or an operand equal to zero 0 indicating a null (end) link. The MCP initializes the Next IOCB Link to zero or to a valid Descriptor pointing to another IOCB.

Word ten, the MLIP Current Data Area Pointer, contains a present, unpaged, indexed EBCDIC string or Word DD pointing to where the data transfer will begin (or resume). The Data Area Pointer is initialized by the MCP and is updated by the MLIP as each block of data is transferred. The format of the Descriptor must agree with the Word/Character Oriented Transfer bit of the MLIP control field; that is, word-oriented I/O requires a word DD while character-oriented I/O requires a string DD. If no data is to be transferred, this word may be an operand equal to zero. If the I/O is in a backwards direction and is character-oriented, the Current Data Area pointer must be initialized to the beginning of the first character beyond the end of the I/O area.

Word eleven, the MLIP Current I/O Length, contains an integer value representing the amount of data left to be transferred. The MCP initializes the length to the amount of data to be transferred (which must be non-negative and less than 2^{20}). The value may represent words or bytes, depending on the Word/Character-Oriented Transfer bit in the MLIP control field. The MLIP decrements the length as data is transferred. When the value is in terms of words, the MLIP will decrement the count by the number of full words of data transferred. If either the Input or Output bits of the MLIP control field are true and the length is initially 0, the

MLIP will terminate the I/O without connecting to the DLP. No error will be reported in this case. The MLIP normally terminates data transfer when the value of the length becomes 0. However, if the Continue Count at the End-of-Length bit of the MLIP control field is on during an input operation, the MLIP will not terminate the I/O; instead, it will continue to accept data from the DLP, but it will not store the data in main memory. The MLIP Current I/O Length will be negative in this case. This feature exists so that the true length of a physical tape record can be discovered (usually for the use of tape parity retry).

Word twelve, the MLIP State and Result, contains a formatted operand of sixteen bits of MLIP/MLI state and thirty-two bits of MLIP result information. The word must be initialized to zero by the MCP. Figure 6-10 is representation of a MLIP State and Result word format.



MV4714

The MLIP Result field contains the following bits:

[0:1] Exception
This bit is the OR of bits 5:5.

[1:1] Attention
This bit is set if the Software Attention bit is set in the MLIP control field.

[2:1] DLP error
This bit is set if any bit is on in the first 48 bits of the DLP result Descriptor after ANDing the logical complement of the Result Mask.

[3:1] MLIP/MLI error
This bit is set if any MLIP/MLI error (bits 21:16) is set.

[4:1] MLiP/Hardware error
This bit is set for other errors detected by the MLIP. The error type, as well as any parameters related to the error will be reported in an Error IOCB

[5:1] Completed after queue suspended
This bit is set if the I/O finished while the Command Queue was marked as suspended.

Bits [21:16] represent the MLIP/MLI error field. Included errors are:

[6:1] Memory protect
This bit is set while transferring data only when the Memory Override bit of the MLIP control field equals zero (memory protect) and when either of the following occurs:

1. on output a word with an odd tag (bit 48=1) is read or;
2. on input an attempt is made to write over a word with an odd tag (bit 48=1).

Figure 6-10. MLIP State and Result Word Format

This bit is also set if the MLIP finds a protected word while reading the DLP command or writing the DLP result. Although this result is most likely an MCP error, the DLP will be left in an unknown state and the "MLIP hung the DLP" bit (19) will also be turned on.

[7:1] Count error
This bit is set if the Input or Output bit is set in the MLIP control field, the MLIP Current I/O Length is not equal to zero when the I/O finishes, and the Ignore Count Error bit of the MLIP command is not set.

[11:4] IOCB index
If the Improper IOCB Word bit (12) is on, this field contains the word index in the IOCB.

[12:1] Improper IOCB word
This bit is set if the MLIP discovers errors in the format of an IOCB word; for example, an incorrect tag.

[13:1] Invalid MLIP control field
This bit is set if the MLIP discovers inconsistent control bits; which are, tag transfers with character-oriented I/O.

[14:1] MLI vertical parity error
This bit is set if the MLIP detects a parity error on the MLI.

[15:1] MLI LPW error
This bit is set if the MLIP detects an incorrect LPW on the MLI.

[16:1] Unexpected DLP status
This bit is set whenever the DLP presents the MLIP with an unexpected status. The MLIP will disconnect, leaving the DLP hung.

[17:1] Non-present DLP
This bit is set whenever the DLP address references a non-present DLP.

[18:1] DLP busy
This bit is set whenever the DLP shows a busy status while attempting to initiate the command and the Command Queue cannot be linked horizontally.

[19:1] MLIP hung the DLP
This bit is set if the reaction of the MLIP to a DLP error is an attempt to hang the DLP.

[20:1] MLI time out
This bit is set if the MLIP times out an operation to the DLP. The MLIP will wait eight milliseconds for a strobe to be issued by a DLP. If a base busy condition is encountered during a connection sequence, the MLIP will wait two seconds for the base to go not busy before causing a timeout error.

[21:1] Invalid MLIP command
This bit is set if the operation is an MLIP command and the command value is undefined.

[27:6] Reserved
This field must be zero.

[31:4] DLP status
If the Unexpected DLP Status bit [16:1] is on, or the command was "read DLP status", this field will contain the value of the DLP status.

Figure 6-10. MLIP State and Result Word Format (Cont)

The MLIP State field is used by the MLIP to record the information required to continue communication with the DLP. The format and content of this information may differ on the different implementations of the MLIP.

Word thirteen, the I/O Start Time Word, is initialized to an operand by the MCP. When the MLIP activates the IOCB, it stores the value of the Time of Day register into this word.

Word fourteen, the I/O Finish Time Word, is initialized by the MCP to be an operand. When the IOCB completes, the MLIP copies the value of the Time-of-Day register into this word.

The Time-of-Day register is visible to both the CPU and the MLIP. If the clock is reset during the time that an I/O operation is taking place, this accounting may be incorrect. The MCP will be responsible for any adjustment required by resetting the Time-of-Day register.

Additional words beyond these may be used by the MCP for MCP purposes. The MLIP will not access any of these additional words.

Command Queue Headers

The Command Queue Header is a 5-word structure used by the MLIP to maintain the current state of a Command Queue. At the request of the MCP, by way of the CUIO operator, the MLIP enqueues I/O commands in a Command Queue and then with the cooperation of an DLP initiates them.

The MLIP can add IOCBs to either the head or the tail of the Command Queue. When the MLIP adds an IOCB (or a chain of IOCBs) to the head of the queue, the chain of commands is followed by way of the Next IOCB Link of IOCBs until the last IOCB is found. Then the MLIP puts the Head IOCB Link of the Command Queue Header in the Next IOCB Link of last IOCB and places the IOCB Self-pointer of the first IOCB into the Head IOCB Link. If the Head IOCB Link was zero (the Command Queue was originally empty), the Tail IOCB Link must also be set using the Self-pointer of the last IOCB. When the MLIP adds an IOCB (or chain of IOCBs) to the tail of a Command Queue, the MLIP must again link down the chain of commands and find the new end IOCB. Once this is done, the IOCB Self Pointer of the first of the new IOCBs is placed in the Next IOCB Link of the last IOCB already in the queue (found by the Tail IOCB Link). Then the Tail IOCB Link is updated with the Self Pointer of the last IOCB that was found by the MLIP linking down the chain of commands the MLIP had been given to enqueue. Again, if the queue was empty, the Head IOCB Link must also be initialized using the self pointer of the first IOCB in the chain.

The MLIP can only remove an IOCB from the head of the Command Queue. When the MLIP removes an IOCB from the queue, the Next IOCB Link is copied from the IOCB into the Head IOCB Link in the Command Queue Header, and an operand with a value of 1 is written into the Next IOCB Link of the IOCB. If the Next IOCB Link was already 0 (the last command in the chain), the Tail IOCB Link in the Command Queue Header is set to 0.

The format of the Command Queue Header is shown is Figure 6-11.

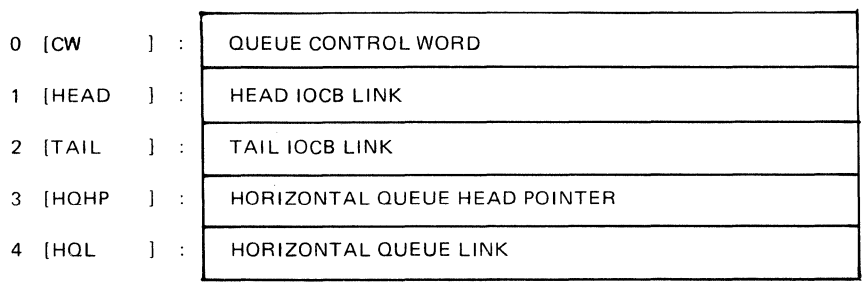
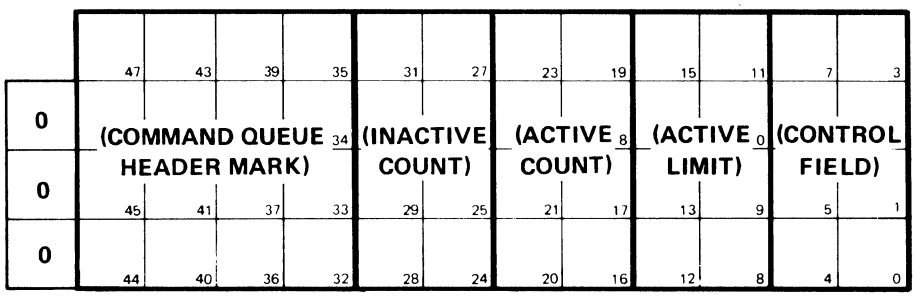


Figure 6-11. Command Queue Header Format

Figure 6-12 shows the Command Queue Header Mark – 10CC. Word zero, the Queue Control Word, is a formatted operand.



MV4716

Queue Control Word Control Field bits are as follows:

[0:1] Suspended
If this bit is on, the MLIP will only initiate an enqueued command if the Immediate bit is on in the MLIP control field. This bit is turned on by the MLIP whenever an I/O from the queue finishes with an error, or whenever an IOCB command finishes while the global MLIP Suspend All Queues flag is true and the Ignore Suspend All Queues flag in the control field of the IOCB is false. The Suspended bit can be turned off by the MCP using an MLIP command.

[1:1] Waiting
If this bit is on, the Command Queue has been dynamically linked into the Horizontal Queue.

[2:1] Horizontal Queue Present
If this bit is on, the Command Queue can be put into a Horizontal Queue. This bit must be consistent with the Horizontal Queue Head Pointer.

[7:5] Reserved (Must be zero)

The Active Limit is a non-zero count of the maximum number of IOCBs which can be active in the queue.

The Active Count is a count of the number of IOCBs which are currently active in the queue.

The Inactive Count is a count of the number of IOCBs which are in the queue but not active (such as, pending).

The MCP initializes the Horizontal Queue Presence bit, the Active Limit, and the Command Queue Header Mark to their proper values; the MCP initializes all other bits to 0.

Figure 6-12. Command Queue Header Mark – 10CC

Word one, the Head IOCB Link, is either zero (indicating a null queue) or a present, unpaged, unindexed word DD pointing to the first IOCB in the queue. The MCP must initialize the Head IOCB Link to zero.

Word two, the Tail IOCB Link, is either zero (indicating a null queue) or a present, unpaged, unindexed Word DD pointing to the last IOCB in the queue. The MCP must initialize the Tail IOCB Link to zero.

Word three, the Horizontal Queue Head Pointer, is either zero (if the Command Queue cannot be part of a Horizontal Queue) or a present, unpaged, indexed Word DD pointing to the Horizontal Queue Head in the Horizontal Queue Array. The MCP initializes this word, which is never altered by the MLIP.

Word four, the Horizontal Queue Link, contains either a zero or a present, unpaged, unindexed Word DD. This word is used to dynamically link Command Queues in the Horizontal Queue. The MLIP is responsible for maintaining this word, but the MCP must initialize it to zero.

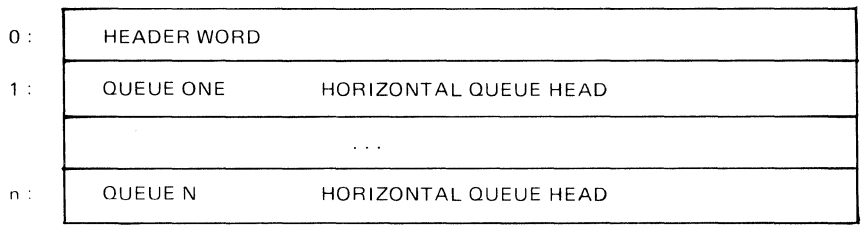
Horizontal Queue Heads/Horizontal Queue Array

The Horizontal Queue Array is an array consisting of a header word and any number of Horizontal Queue Heads. This array is used to associate Command Queues which share a common path (DLP) and have had commands blocked by a busy DLP. This enables the MLIP to do horizontal (path) queuing. The Horizontal Queue Heads are referenced by the Horizontal Queue Head Pointer in each Command Queue Header.

Whenever the MLIP attempts to initiate a command from a Command Queue which has a reference to a Horizontal Queue Head and the DLP is busy, the MLIP will, if necessary, enqueue the Command Queue in the Horizontal Queue. To maintain the Horizontal Queue in First-in First-out (FIFO) order, the MLIP will link down the queue and add the Command Queue to the tail. Whenever the MLIP completes an I/O command and can initiate no more new commands from the Command Queue, it will check the Horizontal Queue referenced by the Command Queue. If the Horizontal Queue has Command Queues linked into it, the MLIP will work down the Horizontal Queue, initiating the appropriate number of commands from each of the Command Queues that are horizontally linked, and will remove (de-link) the command queues from the horizontal queues.

The header word is an operand that is not looked at by the MLIP and will be formatted by the MCP to meet the requirements of memory management. The header word may consist of a "queue mark" in [47:16] (hex 10CE) and a queue array length in [19:20]. Each Horizontal Queue Head in the array is either a zero (indicating an empty queue) or a present, unpaged, unindexed Word DD which points to the first Command Queue Header in the Horizontal Queue. The MLIP maintains the Horizontal Queue Heads, but they must be initialized to zero by the MCP.

The format of the Horizontal Queue Array is shown in Figure 6-13.



MV4717

Figure 6-13. Horizontal Queue Array

Figure 6-14 is a representation of the Horizontal Queue Header Word.

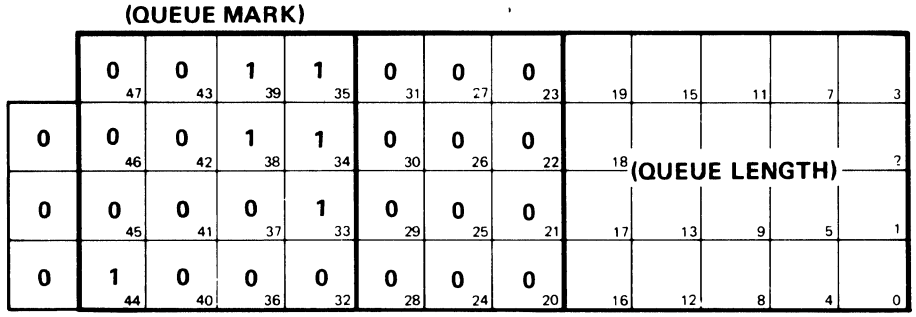


Figure 6-14. Queue Mark – 10CE – Horizontal Queue Header Word

Result Queue Heads/Result Queue Array

The Result Queue Array is an array consisting of a header word and any number of Result Queue Heads. The header word is an operand that is not looked at by the MLIP. It will be formatted by the MCP to meet the requirements of memory management. It may consist of a "queue mark" in [47:16] (hex 10CF) and a queue array length in [19:20]. Each Result Queue Head in the array is either a zero (indicating an empty queue) or a present, unpaged, unindexed Word DD pointing to the first IOCB in the Result Queue. The Result Queue Array is created by the MCP, which initializes the header word and sets all the Result Queue Heads to zero.

The Result Queue is a Last-in First-out (LIFO) queue of completed I/O commands. The Result Queue Head points to the last completed IOCB and the Next Link in the IOCB points to the next to the last, continuing to completion of the queue. The MLIP adds an IOCB to a Result Queue by exchanging the value in the Result Queue Head with a copy of the Self-pointer of the IOCB and writing the original value of the Result Queue Head into the Next Link word of the IOCB. The MCP can empty a Result Queue by "readlocking" the Result Queue Head with a zero. The Descriptor returned by the readlock points to the first IOCB in the Result Queue. (To guarantee correctness, the MCP must wait until the Next Link word of the first IOCB in the queue does not contain an operand with a value of 1.

The format of the Result Queue Array is shown in Figure 6-15.

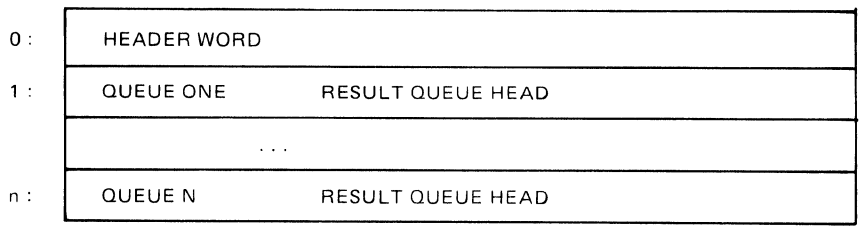


Figure 6-15. Result Queue Array

Figure 6-16 is the Result Queue Header Word format.

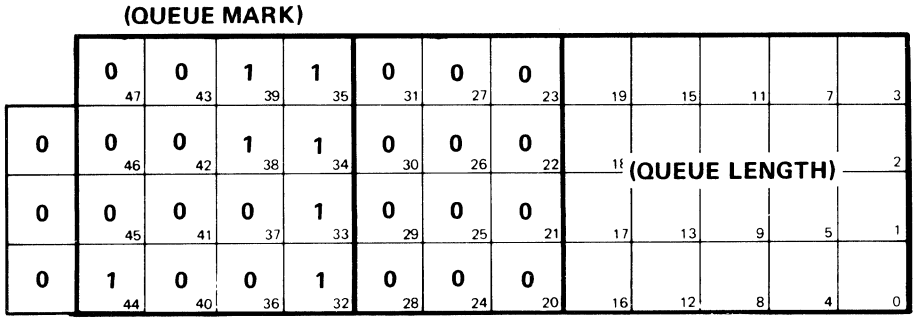


Figure 6-16. Queue Mark – 10CF – Result Queue Header Word

Control of Data Structures

The MLIP cannot create memory structures. The MCP is responsible for allocating space in memory for each structure. Additionally, the MCP is responsible for initializing static conditions in the Command Queue Header (active limit, horizontal queue present). The MCP is also responsible for initializing the dynamic links in all Command, Horizontal, and Result Queues to zero. When an IOCB is created, the MCP is responsible for initializing it.

The MCP initiates an I/O by executing the CUIO operator, which transfers the address of the IOCB that the MCP wants executed to the MLIP. It is the responsibility of the MLIP to enqueue the IOCB into the Command Queue, initiate the operation, and when the operation is done, enqueue the IOCB in the Result Queue. Once the MCP has initiated an IOCB (or chain of IOCBs), the MCP can alter that IOCB (or chain) only in well-defined circumstances, as follows:

- If the IOCB has been returned to the MCP in a Result Queue.
- If the IOCB has been returned to the MCP as a result of a "return queue" MLIP command.
- If the MCP is doing necessary clean up following these commands:
  A "clear DLP" MLIP command.
  A cancel command to a DLP.

The MCP can alter or move a Horizontal Queue Head only when there are no "active" references to it. This case applies when all the Command Queues that reference the Horizontal Queue Head have no outstanding active commands and no enqueued inactive IOCBs. The MCP can alter or move a Command Queue Header only when it has no outstanding active commands and it is not linked into the Horizontal Queue.

MLIP/MEMORY INTERFACE

The MLIP is considered to be an independent memory requestor. All the memory areas shared between the MCP and the MLIP are referenced by way of valid Data or String Descriptors. The MLIP uses Data Descriptors which are incremented or decremented serially; however, the MLIP never creates a Data Descriptor from an absolute address or converts an unindexed Data Descriptor into an indexed one.

MLIP/DLP INTERFACE

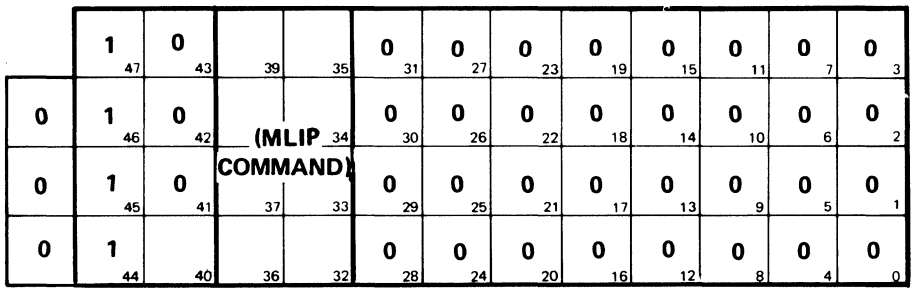
All information sent to the UIO subsystem by the MLIP, except for the Descriptor link, comes from the IOCB or arrays referenced by Descriptors in the IOCB. The Descriptor link, comprised of a host return field and the absolute address of the IOCB, are built by the MLIP and are invisible to the MCP. The MLIP will create the host return field such that its host identification corresponds to the processor ID of its CPU.

The MLIP considers any Descriptor link that is returned by an DLP which does not correspond to a valid IOCB as an invalid link and takes the appropriate actions to inform the MCP of the error.

MLIP COMMANDS

The MLIP/DLP Command bit in the MLIP control field tells the MLIP whether or not the MLIP should execute the command or pass the command on to a DLP. If the MLIP/DLP command bit is true, the MLIP accesses the first word pointed to by the DLP I/O Command Pointer and interprets it.

The format of the MLIP command word is shown in Figure 6-17.



MV4721

Figure 6-17. MLIP Command Word

Valid MLIP Commands, as illustrated in figure 6-17, are (represented in hexadecimal data):

01 Wait for error (Error IOCB)
This command indicates that the IOCB is an "Error" IOCB. The MLIP stores the absolute reference to the IOCB and completes the I/O whenever the MLIP encounters an error that cannot be directly related to or described in another IOCB.

02 Clear DLP
The MLIP performs a "selective clear" of the DLP specified in the DLP Address Word of the IOCB and then places the IOCB in the Result Queue.

03 General Clear
The MLIP performs the MLI master clear handshake on all MLI PORTS and then places the IOCB in the Result Queue.

04 Set the Suspend All Queues flag
The MLIP sets the Suspend All Queues flag and then places the IOCB in the Result Queue. (See the discussion of the Suspended bit in the Command Queue Header Queue Control Word.) The MCP normally requests that the MLIP set the Suspend All Queues flag at the beginning of a memory dump to prevent the initiation of further I/Os.

05 Reset the Suspend All Queues flag

The MLIP resets the Suspend All Queues flag and then places the IOCB in the Result Queue. The MCP normally requests that the MLIP reset the Suspend All Queues flag at the end of a memory dump.

06 Read DLP status

The MLIP connects to the DLP specified in the DLP Address Word of the IOCB and read its status, which is returned in the MLIP State and Result word of the IOCB. The MLIP then places the IOCB in the Result Queue.

07 Activate queue (resets the Suspended bit of a Command Queue)

The MLIP accesses the Command Queue Header referenced by the Command Queue Header Pointer and resets the suspended bit. The activate queue IOCB is added to the Result Queue referenced by its Result Queue Head Pointer. If appropriate, the MLIP will initiate the first IOCB in the activated Command Queue.

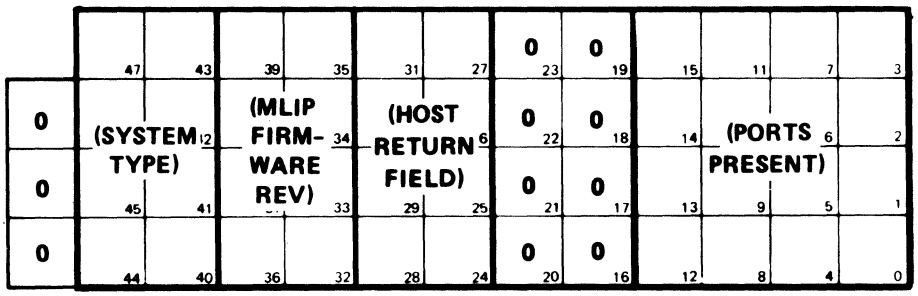
08 Return queue

The MLIP accesses the Command Queue Header referenced by the Command Queue Header Pointer and replaces the Head IOCB Link and the Tail IOCB Link with zeros. The original Head IOCB Link is put in the first word of the area pointed to by the DLP I/O Result Pointer. The return queue IOCB is then added to the Result Queue. The MLIP does not remove the Command Queue from the Horizontal Queue, if currently linked. The Inactive Count field of the Command Queue Header is set to zero.

09 Read MLIP status

The MLIP returns one word of status information, the format of which is shown below, in the first word of the area pointed to by the DLP I/O result pointer.

Figure 6-18 shows the MLIP status word.



MV4722

System Type
This field identifies the type of MLIP and is always a hexadecimal 02.

MLIP Firmware Revision
This field identifies the level of the MLIP firmware.

Host Return Field
This field contains the Host Return Field sent by the MLIP to DLPs. The Host Return Field is a bit vector which uniquely identifies the MLIP to the IODC subsystem.

Ports Present
This field contains a bit vector which identifies which MLI port adapters are present. A bit "on" indicates that the corresponding port is present.

Figure 6-18. MLIP Status Word

0A Discontinue (Error IOCB)

If there is an active Error IOCB, the MLIP causes it to complete with the "discontinued" error code. The MLIP returns one word of result information, the format of which is shown below in Figure 6-19, in the first word of the area pointed to by the DLP I/O Result Pointer.

	47	43	39	35	31	27	23	19	15	11	7	3
	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
	46	42	38	34	30	26	22	18	14	10	6	2
0	0	0	0	0	0	0	0	0	0	0	0	0
	45	41	37	33	29	25	21	17	13	9	5	1
0	0	0	0	0	0	0	0	0	0	0	0	(NE)
	44	40	36	32	28	24	20	16	12	8	4	0

NE: IF THIS BIT IS 0, THE ERROR IOCB WAS DISCONTINUED.
IF THIS BIT IS 1, THERE WAS NO ERROR IOCB TO DISCONTINUE.

MV4723

ne: If this bit is 0, the Error IOCB was discontinued.
If this bit is 1, there was no Error IOCB to discontinue.

In either case, the Discontinue IOCB completes with no errors. Note that a discontinued Error IOCB is handled exactly as any other Error IOCB reporting a valid error.

Figure 6-19. Returned MLIP IOCB Result Descriptor

0B Deactivate Queue (sets the Suspended bit of a Command Queue)

The MLIP accesses the Command Queue Header referenced by the Command Queue Header Pointer and sets the suspended bit. The deactivate queue IOCB is added to the Result Queue referenced by its Result Queue Head Pointer.

ERROR HANDLING

The following chapters discuss the IODC error handling characteristics.

CATEGORIES OF ERRORS

Errors visible to the MLIP can be placed into three categories: errors which reflect only the logical correctness of an I/O, errors which reflect the logical correctness of the I/O and pinpoint a specific failure in the hardware, and errors which cannot be associated with an I/O operation for some reason. The logical correctness of an I/O operation is reported in the MLIP State and Result word of the IOCB associated with the I/O operation. Specifically, bits [4:3] indicate if the MLIP has detected any errors.

The DLP Error bit [2:1] indicates a DLP detected error, which is described in more detail in the DLP result. The MLIP/MLI Error bit [3:1] indicates an error detected by the MLIP; these errors are specified by bits [31:26] of the same word. Errors which are reported by bits 2 or 3 are of the first category.

The MLIP/Hardware Error bit [4:1] indicates that the I/O logically failed because of a hardware failure detected by the MLIP. The exact nature of the failure is not reported in the IOCB associated with the I/O operation; rather, the failure is returned in the result Descriptor of an IOCB reserved for that purpose (the "Error" IOCB). All the IOCB reports is the status of the I/O. The requesting process only needs to know that the I/O failed; it does not need to know how it failed. For example, a failure because of a read data multiple-bit error at address X. That information is useful to the maintenance logging procedure of the MCP (which does not need to know much about what I/O operation caused the hardware failure); therefore, the information is separated into two IOCBs. This eliminates the necessity of reserving many words in the IOCB for storing hardware error parameters.

The third category of errors includes the "Invalid Descriptor Link" and the "Invalid Queue Word." The MLIP does not have any IOCB in which to report these errors, therefore, the Error IOCB is used.

ERROR IOCB/ERROR IOCB COMMAND QUEUE

The Error IOCB and the Command Queue with which it is associated are designed to handle the communication of error information from the MLIP to the MCP. However, because of the special nature of the Error IOCB, both the MLIP control options in the IOCB and the Control Word of the Command Queue must be constructed taking the following precautions.

The MLIP has only one address register into which to store the reference to the Error IOCB. As a result, the MLIP has an Active Limit of one for the Command Queue of the Error IOCB. The MLIP will hang, after an attempt to communicate with the maintenance processor, whenever it has an error to report and does not have a reference to an Error IOCB. Therefore, the MCP will have preferably more than one Error IOCB enqueued in the Command Queue. The MCP must set the Ignore Suspend All Queues bit in all Error IOCBs to prevent the suspension of the Command Queue in the event that the Suspend All Queues flag is on.

Whenever the MLIP uses an Error IOCB, it will not set the DLP Error, the MLIP/MLI Error, the MLIP/Hardware Error, or the Completed After Queue Suspended bits. As long as the MCP has not set the Attention bit in the MLIP control field, the MLIP has no reason to set the Exception bit in the MLIP State and Result word; consequently, the completion of the Error IOCB will not suspend its Command Queue, which allows the MLIP to then initiate a new Error IOCB out of the queue. The Cause I/O Finish Event bit of the MLIP control field of the IOCB must be set for the MCP to ensure that the MLIP will raise the I/O finish interrupt line.

Error IOCBs must be able to store the information that the MLIP generates. The "DLP" I/O Result Pointer reference an area that is currently, 32 words long. This reference is MCP dependent and can be updated in later versions. The format of the result is also MCP dependent. The girst word contains the System Independent Parameter Word. The following nine words contain the contents of the X, B, Y, Temp 1, Temp 2, Temp 3, and Q registers and the C 0, and C 1 8-bit counters, respectively. Because the operation is considered to be a test, the MLIP Current Data Area Pointer can be an operand and the MLIP Current I/O Length can be initialized to zero.

The MLIP uses the Error IOCB whenever it discovers an error in enqueueing or dequeueing an IOCB in a Command Queue, in traversing a Horizontal Queue, in a memory reference, in verifying one of the references of the IOCB to other structures (the Command Queue Pointer, the Result Queue Pointer, or the Next Link), or when receiving a bad Descriptor link from a DLP. (Errors found with the DLP Command Pointer, the DLP Result Pointer, or the MLIP Current Data Area Pointer are reported as an "Improper IOCB Word" error, with the "IOCB index" set in the MLIP State and Result word of the IOCB with the error.)

The information returned by the MLIP in the Error IOCB differs with the type of errors; however, all errors that can be associated with another IOCB will have that IOCB identified in the Error IOCB. An enqueueing error always returns the absolute address of the IOCB that was referenced by the CUIO operator (even though the error may have been discovered while following the links of an IOCB linked behind the first IOCB) and also the address of the offending word. Errors found while verifying the integrity of queue structures may return the absolute address that points into the middle of the structure if one of the elements of the structure is in error. Only errors that occur while transferring data to or from memory, which will have the address of the error and the error type placed in an Error IOCB, will also have an error indication in the original IOCB.

ERROR IOCB RESULT PARAMETERS

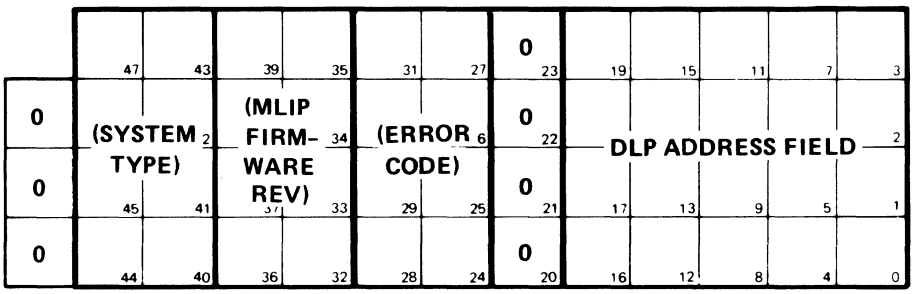
The result area of the Error IOCB contains error parameters that describe the error that occurred and give the context within which the error occurred. These error parameters are of two types: system independent error parameters, and system dependent error parameters.

System independent error parameters provide the MCP with information as to the type of error that occurred, and with any information necessary for error recovery.

System dependent error parameters provide information not required by the MCP but used for logging purposes. The format of these parameters varies from one system to another.

System Independent Parameters

The first result word is a system independent error parameter. This word tells the MCP whether the error is a logging type error, or one that may require MCP action. If MCP recovery action is required (for example, as in the case where a DLP sends an invalid Descriptor Link), then a DLP ADDRESS will be provided. Word zero of ERROR IOCB result information contains the following fields as shown in Figure 6-20.



MV4724

System type
This field identifies the type of MLIP that detected the error. This is always represented by a hexadecimal 02.

MLIP Firmware Revision
This field contains the revision number of the MLIP firmware. Its value may dictate how the system dependent information is to be decoded.

Error Code
This field indicates the type of error being reported. The error codes are in hexadecimal as follows:

01	Invalid Descriptor Link
02	Descriptor Link Parity Error
03	Descriptor Link LPW Error
04	DLP Address Mismatch
05	Host Return Field Mismatch
06	GPW Parity Error
07	Poll Request All Port Timeout
08	Connection Timeout Error
09	Nonexistent Request Error
10	Queuing Error
20	Memory/Hardware Error
40	Error IOCB Discontinued
80	Multiple Completion Attempted

DLP Address Field
This field contains the DLP address for Error Codes hexadecimal 01 through 0F.

Figure 6-20. ERROR IOCB Result

System Dependent Parameters

From word 1 to 31 of the result contains information in the required format defined in Error IOCB/Error IOCB Command Queue heading. This format is dependent on the particular system that generates it. The system type field and MLIP revision field determine the format currently defined.

MLIP ALGORITHMS

The following chapters discuss algorithms performed by the MLIP.

MLIP IDLE LOOP

The MLIP waits for requests from two sources: the CPU through the CUIO operator, and the DLP subsystem through the MLI interface. DLP requests cannot originate without first having received commands from the MLIP through IOCBs.

MLIP CUIO

The CUIO operator in the CPU passes an address to the MLIP. Once the MLIP receives the address, the CUIO operator may be completed. The address received by the MLIP is passed to the Enqueueing procedure. The Enqueueing procedure verifies all links in the chain passed to the MLIP, verifies the Command Queue Head or Tail Pointers as required, and upon detecting no errors, places the chain either at the head or tail of the Command Queue, depending upon the state of the Queue at Head bit of the Control Word of the first IOCB in the chain. If enqueueing is completed successfully, the MLIP attempts to initiate IOCBs from the Command Queue in which the IOCB chain was enqueued, according to the criteria explained in the discussion under the heading Initiate Queue.

ENQUEUEING AN IOCB CHAIN IN A COMMAND QUEUE

The IOCB chain to be enqueued (which may contain only one IOCB) is checked to ensure that the chain is correct. The MLIP checks that each IOCB in the chain contains an IOCB Mark in the Control Word of each IOCB, and that each Next IOCB Link is a valid Descriptor which points to an IOCB. This process is repeated until the MLIP finds the last IOCB in the chain. The last IOCB in the chain contains a null Next IOCB Link (tag=0). If the chain is found to be correct, the MLIP evaluates the Descriptor in the Command Queue Head Pointer word of the first IOCB in the chain and ensure that the reference is a valid Descriptor. If the Descriptor is valid, the Queue Mark in the Command Queue Header Control Word is checked to ensure that it is indeed a Command Queue.

If all links, Descriptors and marks have been found to be correct, enqueueing may proceed. If the Command Queue Head Pointer is null (tag=0), the chain becomes the queue. The Command Queue Head Pointer is replaced by the Self-pointer of the first IOCB in the chain to be added, and the Command Queue Tail Pointer is replaced by the Self Pointer of the last IOCB in the chain to be added, and the enqueueing is completed. If the queue is not empty, the IOCB chain is placed at the head of the queue if the Queue at Head bit of the Control Word of the first IOCB in the chain is set, and at the tail of the queue if the Queue at Head bit is not set. To place the chain at the head of the queue, the MLIP replaces the Next IOCB Link word of the last IOCB in the chain with the contents of the Command Queue Head Pointer. The Command Queue Head Pointer is then replaced by the Self-pointer of the first IOCB in the chain. The enqueueing is then completed. To place the chain at the tail of the queue, the Command Queue Tail Pointer is evaluated (if it is a valid Descriptor) and the IOCB to which it points is verified to contain an IOCB Mark in the IOCB Control Word. If these criteria are met, the chain is linked into the tail of the Command Queue by first replacing the contents of the Next IOCB Link word of the tail IOCB in the queue with the Self-pointer of the first IOCB in the chain. Then the Command Queue Tail Pointer is replaced by the Self Pointer of the last IOCB in the chain to be added. The enqueueing is then complete. At this point, the MLIP increments the Inactive Count field of the Command Queue Control Word by the number of IOCBs added to the chain.

If any of the validity checks performed failed, the chain is not placed into the Command Queue. The Error IOCB is used to report an enqueueing error, and to report the absolute address of the first IOCB in the chain to be added, and the absolute address of the word that failed to pass a validity check.

INITIATE QUEUE

The Control Word of the Command Queue to be initiated is first verified to contain a Queue Mark. If it does not, the queue cannot be initiated, and the Error IOCB is used to report the error. Next, the Head IOCB Link word of the Command Queue is referenced. If it is null (tag=0), there is nothing more to do in this queue. If the Head IOCB Link word is a valid Descriptor, the Control Word of the IOCB is accessed. If the Control Word contains an IOCB Mark, the MLIP Control field is interpreted. If the Immediate bit in the MLIP control field is not on, and the Active Count field is not less than the Active Limit field of the Command Queue Control Word, or the Suspended bit of the Command Queue Control Word is on, there is nothing to do in this queue. (The Suspend All Queues flag does not cause a queue to be suspended during the initiation of the queue, but only when an IOCB is added to a result queue.)

If the MLIP Control field indicates that the IOCB is a DLP command (MLIP/DLP Command bit equal to zero) and the Waiting bit in the Command Queue Control Word is one, this queue is known to be linked into a horizontal queue and there is nothing more for the MLIP to do in this queue or in the horizontal queue structure. If the MLIP/DLP Command bit indicates a DLP command, and the queue is not marked waiting, the MLIP attempts to connect to the DLP specified in the DLP Address Word. If the connection is established, and the DLP is busy, the MLIP attempts to place the Command Queue in a horizontal queue (see explanation on adding to a horizontal queue). If the DLP is not present, a Non-Present DLP Error is stored into the IOCB MLIP State and Result word, and the IOCB is taken out of the queue and placed into the result queue. (A busy DLP indicates a "Disconnect" status.)

If the DLP is present and attempting to send a Descriptor link to the MLIP, the MLIP must save the address of the command queue header, turn the line around and handle the DLP request (see Poll Request/Turnaround description), restore the saved queue header address, and begin the queue initiation process all over again. If the DLP is present, but not in the Idle state, the IOCB is taken out of the Command Queue, an Unexpected DLP Status error along with the DLP status is placed into the IOCB MLIP State and Result word, the IOCB is added to the result queue, and the MLIP disconnects from the DLP. If the DLP is present and in the Idle state, or if the MLIP/DLP Command bit in the MLIP Control field indicates this IOCB is an MLIP command, the IOCB is taken out of the command queue and initiated, and the queue initiation process is followed again until there is nothing more to do in this queue.

When there is nothing more to do in the queue, if there is a horizontal queue (indicated by the Horizontal Queue Presence bit in the Command Queue Control Word), the Horizontal Queue Head Pointer in the Command Queue header is used to reference the Horizontal Queue Head. If the horizontal queue is not empty (an empty queue is indicated by a tag=0 in the horizontal queue head), the first Command Queue in the horizontal queue is removed from the horizontal queue and queue initiation proceeds with this new Command Queue. Command Queues are removed from the horizontal queue and initiated in this manner until the horizontal queue has been emptied or the DLP goes to a non-Idle state after receiving a command Descriptor. At this time, if the MLIP is connected to a DLP, it will disconnect.

REMOVING A COMMAND QUEUE FROM A HORIZONTAL QUEUE

The Command Queue referenced by the Horizontal Queue Head is removed from the horizontal queue by first verifying the Queue Mark in the Command Queue Control Word, and then storing the Command Queue Horizontal Queue Link into the Horizontal Queue Head. The Command Queue Horizontal Queue Link is then set to 0, and the Waiting bit in the Command Queue Control Word is reset to ZERO.

ADDING A COMMAND QUEUE TO A HORIZONTAL QUEUE

If the Horizontal Queue Presence bit in the Command Queue Control Word is not on, the command queue cannot be placed in a horizontal queue, and the IOCB must be taken out of the queue. A DLP Busy error is placed into the MLIP State and Result Word of the IOCB, and the IOCB is placed into the result queue.

The MLIP will then disconnect from the DLP. If the Horizontal Queue Presence bit is on, the command queue is placed into the horizontal queue only if the Immediate bit in the MLIP Control field of the IOCB Control Word is on, or if the Active Count field of the Command Queue Control Word is zero; otherwise, completion of an outstanding IOCB from this command queue will ultimately result in this command queue being initiated. To place the command queue into the horizontal queue, the Horizontal Queue Head Pointer of the Command Queue is used as a reference to the Horizontal Queue Head (after performing a validity check). If the Horizontal Queue Head is null (tag=0), the Horizontal Queue Head is replaced by the Command Queue Head Pointer word of the IOCB.

If the Horizontal Queue Head is not null, the last Command Queue in the Horizontal Queue is found by verifying the Queue Marks of the Command Queues in the Horizontal Queue and evaluating references to the next Command Queue in the chain using the Horizontal Queue Link pointer word in the Command Queue Header (after ensuring that it is a valid Descriptor). The Horizontal Queue Link of the last Command Queue in the Horizontal Queue is null (tag=0). If the tail of the Command Queue was successfully located, the new Command Queue is added by replacing the Horizontal Queue Link word of the tail Command Queue with the Command Queue Head Pointer of first IOCB in the Command Queue to be added to the Horizontal Queue. The Horizontal Queue Link word of the Command Queue being added is then set to zero. The horizontal queuing is then completed. If any of the validity checks failed, the error is reported through the Error IOCB. The error parameter indicates an enqueueing error, the absolute address of the Command Queue that was to be placed into the horizontal queue, and the absolute address of the word that failed a validity check. The Command Queue cannot then be added to the Horizontal Queue.

ATTEMPTING TO CONNECT TO A DLP

An MLIP/Distribution Card handshaking procedure is necessary to establish connection to a DLP. During the connection protocol, several error conditions may arise, such as a timeout, indicating that no base is there (a DLP address parity error indication, or a port busy indication). In the case of timeouts and address parity errors, the IOCB must be dequeued, the appropriate error is marked in the MLIP State and Result word of the IOCB, and the IOCB is added to the result queue. A port busy condition arises when another system is accessing the same base through another distribution card.. The MLIP response to port busy is simply to wait for the DLP connection to be established as soon as the base becomes not busy. When the connection appears to have been established after a port busy, if the DLP STC=10, the connection was not established, the port is still busy, but another base attached to a LEM is requesting MLIP attention. The MLIP response is to disconnect from the busy base, save the queue head address (for a DLP command) or the IOCB address (for an MLIP command), handle the Poll Request from the requesting base, restore the address, and attempt to reinitiate the operation.

DEQUEUEING AN IOCB FROM A COMMAND QUEUE

If the Don't Count bit of the MLIP Control field of the IOCB Control word is zero, the Active Count field of the Command Queue Control Word is incremented by one. The Inactive Count field of the Command Queue Control Word is decremented by one. The IOCB is dequeued by replacing the Command Queue Head IOCB Link with the contents of the IOCB Next IOCB Link word. The Next IOCB Link word of the IOCB is then replaced by an operand with a value of one, to mark the IOCB as "in process". If the Command Queue Head IOCB Link is now null (tag=0), the Command Queue Tail IOCB Link is made null by storing into it a zero.

INITIATING AN IOCB

The Time-of-Day register is stored into the IOCB I/O Start Time Word. If the MLIP/DLP Command bit of the MLIP Control field of the IOCB Control Word indicates this IOCB is an MLIP command, the MLIP performs the indicated operation (see the explanation of MLIP operations). If this IOCB contains a DLP command, the MLIP performs certain validity checks of the various control bits and variables. In particular, if either the Input or Output bit of the MLIP Control field is on, the following tests are performed:

1. If the MLIP Current I/O Length word of the IOCB is zero, nothing is sent to the DLP and the IOCB is placed into the result queue with no DLP errors and no data transferred.
2. If the transfer is flagged to be character-oriented by the Word/Character-Oriented Transfer bit of the MLIP Control field of the IOCB Control Word, the MLIP Current Data Area Pointer is an EBCDIC string DD and the Tag Control field of the MLIP Control field of the IOCB Control Word requires tags to be either 0, 2, or 3. If the transfer is flagged to be word-oriented by the Word/Character Oriented Transfer bit of the MLIP Control field of the IOCB Control Word, the MLIP Current Data Area Pointer is a valid Word DD and the Tag Control field may indicate any of the force tag options or one of the transfer tag options.

The MLIP must now do any initialization required by its particular implementation to the MLIP State field of the MLIP State and Result word of the IOCB. The DLP command Descriptor can now be sent to the DLP by evaluating the Descriptor in the DLP I/O Command Pointer word of the IOCB. It must contain a valid Descriptor which points to an area of memory where the MLIP finds the DLP command Descriptor. Since the DLP command Descriptors vary in length from DLP to DLP and from operation to operation, the MLIP must know how many characters of command Descriptor to send, as a DLP broken in this state takes all of memory if the MLIP cooperated; hence, the MLIP obtains the command Descriptor length in the DLP Command/Result Lengths word of the IOCB by isolating the length from the appropriate field. Additionally, the command length must be even, as the MLIP sends the DLP command two characters at a time. As the MLIP sends the command Descriptor to the DLP, the command length is decremented. If the length is zero and the DLP requests more command Descriptor data, the MLIP disconnects from the DLP, and places the IOCB into the result queue with an Unexpected DLP Status error (which also returns the DLP status).

After sending the DLP the command Descriptor, the MLIP sends the command Descriptor LPW word, and then generates the Descriptor link using the absolute memory address of the IOCB and the processor ID to generate the host return field, sends the Descriptor link and Descriptor link LPW to the DLP.

GET DLP RESULT

Like the command Descriptor, the DLP result Descriptor is of variable length. Consequently, the MLIP must evaluate the Descriptor in the DLP I/O Result Pointer word of the IOCB to find where to place the DLP result Descriptor. Additionally, the MLIP gets the result Descriptor length from a field in the DLP Command/Result Lengths word of the IOCB. The result length must also be even, as the DLP sends the result Descriptor two characters at a time. As the DLP sends the result Descriptor, the MLIP decrements the length counter. If the length is zero, and the DLP wants to send more result Descriptor characters, the MLIP disconnects from the DLP, and place the IOCB into the result queue with an Unexpected DLP Status error (which also returns the DLP status); otherwise, the first word of the DLP result Descriptor (up to 48 bits) is then ANDed with the complement of the contents of the Result Mask word of the IOCB. If the result of the AND is not zero, the DLP Error bit is set in the MLIP State and Result word of the IOCB. The MLIP reads the result Descriptor LPW and marks any LPW errors in the MLIP State and Result word of the IOCB. The IOCB is then placed into the result queue.

ADDING AN IOCB TO A RESULT QUEUE

Before the IOCB is actually added to the result queue, several housekeeping chores must be performed. First, the Command Queue Head Pointer of the IOCB is evaluated to find the Command Queue Header. Next, the Suspended bit of the Command Queue Control Word is set if the Suspend All Queues flag is set, and the Ignore Suspend All Queues bit in the IOCB Control Word is reset.

If the Suspended bit of the Command Queue Control Word is on, the Completed After Queue Suspended bit of the MLIP State and Result Word of the IOCB is set. If the Attention bit in the IOCB Control Word is on, the Attention bit in the MLIP State and Result Word of the IOCB is also set.

The Count Error bit and the MLIP/MLI Error bit is set in the MLIP State and Result Word of the IOCB if either the Input bit or the Output bit is on in the IOCB Control Word, and if the Ignore Count Error bit is off in the IOCB Control Word and if the MLIP Current I/O Length word in the IOCB is not equal to zero. The Exception bit in the MLIP State and Result Word of the IOCB and the Suspended bit in the Command Queue Control Word is set if any of the bits in the [5:5] field of the MLIP State and Result Word are on.

The Active Count field of the Command Queue Control Word is decremented if the Don't Count bit of the IOCB Control Word is not on. The contents of the Time-of-Day register is placed into the I/O Finish Time Word.

The Result Queue Head Pointer word of the IOCB is evaluated as a reference to the Result Queue Head. The IOCB Self-pointer is stored into the Result Queue Head with a readlock operation, and the previous contents of the Result Queue Head are placed into the IOCB Next Link word.

The MLIP causes an I/O Finish Interrupt in the CPU if the Exception bit in the IOCB MLIP State and Result word is on, the Cause I/O Finish Event bit in the IOCB Control Word is on, the Active Count field of the Command Queue Control Word is zero, the Exception bit of the IOCB MLIP State and Result word is not on, and the Command Queue Head IOCB Link word is null (tag=0) (end of chain condition), or the CPU is executing the Idle operator (optional).

POLL REQUEST/TURNAROUND

Poll Request connects to a DLP that has requested MLIP attention. Turnaround handles a DLP that is already connected to the MLIP and has either Data or Result Descriptor information to send. Poll Request reads the Global Priority Word from the MLI and saves the DLP Address field so the MLIP will know which DLP is connected, and then completes the MLI connection algorithm. Turnaround, since the MLIP is already connected, has the DLP Address available.

Once connected, the DLP first transmits to the MLIP the Descriptor Link and Descriptor link LPW. The MLIP then must verify the Descriptor link to ensure its validity. The Descriptor link is invalid if the LPW was incorrect, the Host Return Field is incorrect, the IOCB address field of the Descriptor link does not reference a word in memory with an IOCB Mark, or the DLP Address Word of the IOCB does not match the DLP Address field of the Global Priority Word (or in the case of Turnaround, the DLP Address of the DLP to which the MLIP is connected). If the Descriptor link is invalid for any of the above reasons, the MLIP disconnects from the DLP (which leaves the DLP "hung") and reports the invalid Descriptor link through the Error IOCB.

If the Descriptor link is valid, the MLIP verifies that the DLP request is proper, as follows:

If the DLP goes to Read Status, the MLIP verifies that the Input bit of the IOCB Control Word is on, and then handle the input burst.

If the DLP goes to Write Status, the MLIP verifies that the Output bit of the IOCB Control Word is on, and then handle the output burst.

If the DLP goes to Send Result Descriptor Status, the MLIP gets the result Descriptor (see explanation on Get Result Descriptor, which adds the IOCB to the result queue); if the DLP is still connected (the MLIP disconnects if any errors are encountered in getting the result Descriptor or in adding the IOCB to the result queue), the MLIP initiates the command queue that the IOCB referenced through the Command Queue Head Pointer.

If the DLP does not go to Read Status, Write Status, or Send Result Descriptor Status, or if the Read Status or Write Status is invalid due to the Input or Output bit values, the MLIP places an Unexpected DLP Status Error into the MLIP State and Result word of the IOCB, place the IOCB into the result queue, and disconnects from the DLP (which will probably leave the DLP hung).

If the MLIP performed an Input Burst or an Output Burst, the MLIP updates its burst parameters in the IOCB and disconnects.

MLIP OPERATIONS

There are various commands upon which the MLIP is capable of acting. These commands are flagged in the IOCB Control Word if the MLIP/DLP Command bit is on. The actual command, however, is found by evaluating the reference contained in the DLP I/O Command Pointer word, which points to a memory area where the MLIP may actually find its command. The MLIP then fetches the command word, and verifies that it is one that the MLIP recognizes. If it is not a valid command, the MLIP Invalid Operation bit and the MLIP/MLI Error bits are turned on in the IOCB MLIP State and Result word, and the IOCB is placed into the result queue.

If the command is in fact a valid one, the MLIP performs immediately the command, as follows:

Test Wait For Error Operation
: The MLIP first checks to see if it already has a pending Wait For Error operation. If not, the MLIP saves the IOCB address, and uses the error IOCB to report various types of errors, when they occur.

Master Clear Operation
: The MLIP first ensures that it is not connected to any DLPs, and disconnect if it is connected. The the MLI Master Clear sequence is implemented, and the IOCB is placed into the result queue.

Set/Reset Suspend All Queues flag Operation
: This operation either sets or resets, as indicated, the Suspend All Queues flag. The IOCB is then placed into the result queue.

Activate Queue Operation
: The Command Queue Head Pointer reference in the IOCB is used to reference a Command Queue. The Queue Mark is then verified, and the MLIP resets the Suspended bit in the Command Queue Control Word. The IOCB is then placed into the result queue. If the Suspend All Queues flag is on at this time, at least the next IOCB in the Command Queue could be initiated, as the command queue is not suspended, even if the Suspend All Queues flag is on, when any MLIP operation is placed into a result queue.

Deactivate Queue Operation
: The Command Queue Head Pointer reference in the IOCB is used to reference a Command Queue. The Queue Mark is then verified, and the MLIP sets the Suspended bit in the Command queue Control word. The IOCB is then placed into the result queue.

Return Queue Operation

The Command Queue Head Pointer reference in the IOCB is used to reference a Command Queue Header. The Queue Mark is then verified. The first word of the area pointed to by the DLP I/O Result Pointer is then replaced by the contents of the Command Queue Head IOCB Link word. Null links (zeros) are then placed into the Command Queue Head IOCB Link word and the Command Queue Tail IOCB Link words. The IOCB is then placed into the result queue.

Clear DLP Operation and Read DLP Status

The MLIP checks that it is not connected to a DLP; if it is connected to the wrong DLP (which should show a status of "Idle" or "Disconnect"), it disconnects. If MLIP is not then connected to the correct DLP, it attempts to connect to the DLP as explained elsewhere. If the connection is successful, and this is a Clear DLP Operation, the DLP is selectively cleared. If this is a Read DLP Status OP, the current DLP status is placed into the DLP Status field of the MLIP State and Result Word of the IOCB. If the connection is not successful, the appropriate error is noted in the MLIP State and Result Word of the IOCB. The IOCB is then placed into the result queue.

DATA TRANSFER

The MLIP considers memory words to be divided into six characters plus a tag. The characters labels are shown in Figure 6-21.

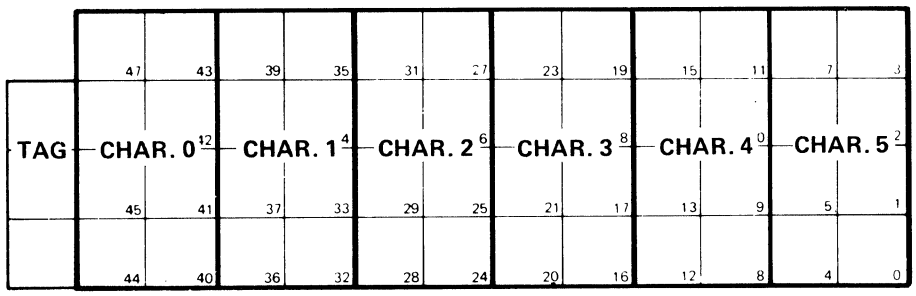


Figure 6-21. Memory Word Characters

For forward operations data transfers are left to right; that is, the first character accessed in the word is the TAG if tag transfer is specified or Character 0 if tag transfer is not specified. Character 5 is the last character accessed. For backward operations, the opposite is true; character 5 is the first character accessed and character 0 (or the tag) is the last.

Data transfer can begin at any character position of a word. If the transfer does not begin at the left hand edge of the word for a forward operation or the right hand edge for a backward operation, the prior contents of the word (which are left of the beginning character for forward or to the right of the beginning character for backward) must not be changed.

Similarly, data transfer can end at any character position. In character-oriented operations, a partial word is filled with the prior contents of the word. In word-oriented operation, partial words are zero filled.

GLOSSARY OF MLIP/IODC OPERATING TERMS

The following are some miscellaneous terms and mnemonics useful in understanding MLIP/IODC concepts:

MLIP
: Message Level Interface Processor – portion of CPU logic which controls operations between the Data Processor and the IODC and its associated DLPs.

IODC
: Input/Output Data Communication – subsystem utilized for I/O and Datacomm operations, common to the MLI interface specifications.

IOCB
: Input/Output Control Block – a contiguous area of memory containing the necessary information for the performance of an I/O or MLIP operation.

CUIO
: Communicate with Universal I/O – a variant mode operator (954C) which starts an operation to the MLIP or IODC using a Data Descriptor found in the top of the stack pointing to the first word of the IOCB.

IOCB MARK
: A value of 4"1OCB" found in [47:16] of the first word in an IOCB used by the logic to verify this is actually the first word of an IOCB.

ERROR IOCB
: An IOCB set aside by the MCP to be used by the MLIP to terminate an I/O operation when normal error termination is not possible.

MLI
: Message Level Interface – a 25 line bi-directional interface between the MLIP and the IODC containing data and control information.

MLIP/CPU INTERFACE
: Connection between CPU and MLIP, primarily Z1 bus, Z5 bus, C register, and micro-module address lines.

MLIP/IODC INTERFACE
: Connection between IODC and MLIP called MLI.

DLP
: Data Link Processor – a specialized micro-processor used to transfer information to and from a peripheral device.

POLL TEST
: Process of MLIP connecting to IODC.

POLL REQUEST
: Process of IODC reconnecting to MLIP following operation initiated by MLIP.

GLOBAL PRIORITY WORD
: A word returned to MLIP during POLL REQUEST indicating priority of each DLP requesting connection to the MLIP.

COMMAND QUEUE
A linking together of IOCBs in the order in which they will be performed.

RESULT QUEUE
A linking together of IOCBs as the I/O operation is completed.

COMMAND QUEUE HEADER
A structure used to maintain the current state of a command queue.

RESULT QUEUE HEADER
A structure used to maintain the current state of the completed I/O operations.

APPENDIX A
OPERATORS

Table A-1. Operators, Alphabetical List

Name	Mnemonic	Hexadecimal Code
ADD	ADD	80
BIT RESET	BRST	9E
BIT SET	BSET	96
BRANCH FALSE	BRFL	A0
BRANCH TRUE	BRTR	A1
BRANCH UNCONDITIONAL	BRUN	A2
CHANGE SIGN BIT	CHSN	8E
COMPARE CHARACTERS EQUAL DELETE	CEQD	F4
COMPARE CHARACTERS EQUAL UPDATE	CEQU	FC
COMPARE CHARACTERS GREATER OR EQUAL DELETE	CGED	F1
COMPARE CHARACTERS GREATER OR EQUAL UPDATE	CGEU	F9
COMPARE CHARACTERS GREATER DELETE	CGTD	F2
COMPARE CHARACTERS GREATER UPDATE	CGTU	FA
COMPARE CHARACTERS LESS OR EQUAL DELETE	CLED	F3
COMPARE CHARACTERS LESS OR EQUAL UPDATE	CLEU	FB
COMPARE CHARACTERS LESS DELETE	CLSD	F0
COMPARE CHARACTERS LESS UPDATE	CLSU	F8
COMPARE CHARACTERS NOT EQUAL DELETE	CNED	F5
COMPARE CHARACTERS NOT EQUAL UPDATE	CNEU	FD
CONDITIONAL HALT (all modes)	HALT	DF
COUNT BINARY ONES	CBON	95BB
COMMUNICATE WITH UNIVERSAL I/O	CUIO	954C
DELETE TOP-OF-STACK	DLET	B5
DISABLE EXTERNAL INTERRUPT	DEXI	9547
DIVIDE	DIVD	83
DYNAMIC BRANCH UNCONDITIONAL	DBUN	AA
DYNAMIC FIELD INSERT	DINS	9D
DYNAMIC FIELD ISOLATE	DISO	9B
DYNAMIC FIELD TRANSFER	DFTR	99
DYNAMIC SCALE LEFT	DSLF	C1
DYNAMIC SCALE RIGHT FINAL	DSRF	C7
DYNAMIC SCALE RIGHT ROUND	DSRR	C9
DYNAMIC SCALE RIGHT SAVE	DSRS	C5
DYNAMIC SCALE RIGHT TRUNCATE	DSRT	C3
ENABLE EXTERNAL INTERRUPTS	EEXI	9546
END EDIT	ENDE	DE
END FLOAT	ENDF	D5
ENTER	ENTR	AB
EQUAL	EQUL	8C
ESCAPE TO 16-BIT INSTRUCTION	VARI	95
EVALUATE	EVAL	AC

Table A-1. Operators, Alphabetical List (Cont)

Name	Mnemonic	Hexadecimal Code
EXCHANGE	EXCH	B6
EXECUTE SINGLE MICRO, SINGLE POINTER UPDATE	EXPU	DD
EXECUTE SINGLE MICRO DELETE	EXSD	D2
EXECUTE SINGLE MICRO UPDATE	EXSU	DA
EXIT	EXIT	A3
EXTENDED MULTIPLY	MULX	8F
FIELD INSERT	INSR	9C
FIELD ISOLATE	ISOL	9A
GREATER THAN	GRTR	8A
GREATER THAN OR EQUAL	GREQ	89
INDEX	INDX	A6
INDEX AND LOAD NAME	NXLN	A5
INDEX AND LOAD VALUE	NXLV	AD
INPUT CONVERT DELETE	ICVD	CA
INPUT CONVERT UPDATE	ICVU	CB
INSERT CONDITIONAL (edit mode)	INSC	DD
INSERT DISPLAY SIGN (edit mode)	INSG	D9
INSERT MARK STACK	IMKS	CF
INSERT OVERPUNCH (edit mode)	INOP	D8
INSERT UNCONDITIONAL (edit mode)	INSU	DC
INTEGER DIVIDE	IDIV	84
INTEGERIZE ROUNDED	NTGR	87
INTEGERIZE TRUNCATED	NTIA	86
INTEGERIZE ROUNDED DOUBLE-PRECISION	NTGD	9587
INVALID OPERATOR (all modes)	NVLD	FF
LEADING ONE TEST	LOG2	958B
LINKED LIST LOOKUP	LLLU	95BD
LESS THAN	LESS	88
LESS THAN OR EQUAL	LSEQ	8B
LITERAL CALL ONE	ONE	B1
LITERAL CALL ZERO	ZERO	B0
LITERAL CALL 8-BITS	LT8	B2
LITERAL CALL 16-BITS	LT16	B3
LITERAL CALL 48-BITS	LT48	BE
LOAD	LOAD	BD
LOAD TRANSPARENT	LODT	95BC
LOGICAL AND	LAND	90
LOGICAL EQUAL	SAME	94
LOGICAL EQUIVALENCE	LEQV	93
LOGICAL NEGATE	LNOT	92
LOGICAL OR	LOR	91
MAKE PROGRAM CONTROL WORD	MPCW	BF
MARK STACK	MKST	AE
MASKED SEARCH FOR EQUAL	SRCH	95BE
MOVE CHARACTERS (edit mode)	MCHR	D7
MOVE NUMERIC UNCONDITIONAL (edit mode)	MVNU	D6

Table A-1. Operators, Alphabetical List (Cont)

Name	Mnemonic	Hexadecimal Code
MOVE TO STACK	MVST	95AF
MOVE WITH FLOAT (edit mode)	MFLT	D1
MOVE WITH INSERT (edit mode)	MINS	D0
MULTIPLY	MULT	82
NAME CALL	NAMC	40 through 7F
NO OPERATION (all modes)	NOOP	FE
NORMALIZE	NORM	958E
NOT EQUAL	NEQL	8D
OCCURS INDEX	OCRX	9585
OVERWRITE DELETE	OVRD	BA
OVERWRITE NON-DELETE	OVRN	BB
PACK DELETE	PACD	D1
PACK UPDATE	PACU	D9
PUSH DOWN STACK REGISTERS	PUSH	B4
READ AND CLEAR OVERFLOW FLIP-FLOP	ROFF	D7
READ PROCESSOR IDENTIFICATION	WHOI	954E
READ PROCESSOR REGISTER	RPRR	95B8
READ TAG FIELD	RTAG	95B5
READ TIME-OF-DAY CLOCK	RTOD	95A7
READ TRUE/FALSE FLIP-FLOP	RTFF	DE
READ WITH LOCK	RDLK	95BA
REMAINDER DIVIDE	RDIV	85
RESET FLOAT (edit mode)	RSTF	D4
RETURN	RETN	A7
ROTATE STACK DOWN	RSDN	95B7
ROTATE STACK UP	RSUP	95B6
RUNNING INDICATOR	RUNI	9541
SCALE LEFT	SCLF	C0
SCALE RIGHT FINAL	SCRF	C6
SCALE RIGHT ROUNDED	SCRR	C8
SCALE RIGHT SAVE	SCRS	C4
SCALE RIGHT TRUNCATE	SCRT	C2
SCALE-IN	SCNI	954A
SCAN-OUT	SCNO	954B
SCAN WHILE EQUAL DELETE	SEQD	954F
SCAN WHILE FALSE UPDATE	SEQU	95FC
SCAN WHILE FALSE DELETE	SWFD	95D4
SCAN WHILE FALSE UPDATE	SWFU	95DC
SCAN WHILE GREATER OR EQUAL DELETE	SGED	95F1
SCAN WHILE GREATER OR EQUAL UPDATE	SGEU	95F9
SCAN WHILE GREATER DELETE	SGTD	95F2
SCAN WHILE GREATER UPDATE	SGTU	95FA
SCAN WHILE LESS OR EQUAL DELETE	SLED	95F3
SCAN WHILE LESS OR EQUAL UPDATE	SLEU	95FB

Table A-1. Operators, Alphabetical List (Cont)

Name	Mnemonic	Hexadecimal Code
SCAN WHILE LESS DELETE	SLSD	95F0
SCAN WHILE LESS UPDATE	SLSU	95F8
SCAN WHILE NOT EQUAL DELETE	SNED	95F5
SCAN WHILE NOT EQUAL UPDATE	SNEU	95FD
SCAN WHILE TRUE DELETE	SWTD	95D5
SCAN WHILE TRUE UPDATE	SWTU	95DD
SET DOUBLE TO TWO SINGLES	SPLT	9543
SET EXTERNAL SIGN	SXSN	D6
SET INTERVAL TIMER	SINT	9545
SET PROCESSOR REGISTER	SPRR	95B9
SET TAG FIELD	STAG	95B4
SET TO DOUBLE-PRECISION	XTND	CE
SET TO SINGLE-PRECISION ROUNDED	SNGL	CD
SET TO SINGLE-PRECISION TRUNCATED	SNGT	CC
SET TWO SINGLES TO DOUBLE	JOIN	9542
SKIP FORWARD DESTINATION CHARACTERS (edit mode)	SFDC	DA
SKIP FORWARD SOURCE CHARACTERS (edit mode)	SFSC	D2
SKIP REVERSE DESTINATION CHARACTERS (edit mode)	SRDC	DB
SKIP REVERSE SOURCE CHARACTERS (edit mode)	SRSC	D3
STORE DESTRUCTIVE	STOD	B8
STORE NON-DELETE	STON	B9
STRING ISOLATE	SISO	D5
STUFF ENVIRONMENT	STFF	AF
SUBTRACT	SUBT	81
TABLE ENTER EDIT DELETE	TEED	D0
TABLE ENTER EDIT UPDATE	TEEU	D8
TRANSFER UNCONDITIONAL DELETE	TUND	E6
TRANSFER UNCONDITIONAL UPDATE	TUNU	EE
TRANSFER WHILE EQUAL DELETE	TEQD	E4
TRANSFER WHILE EQUAL UPDATE	TEQU	EC
TRANSFER WHILE GREATER OR EQUAL DELETE	TGED	E1
TRANSFER WHILE GREATER OR EQUAL UPDATE	TGEU	E9
TRANSFER WHILE GREATER DELETE	TGTD	E2
TRANSFER WHILE GREATER UPDATE	TGTU	EA
TRANSFER WHILE LESS OR EQUAL DELETE	TLED	E3
TRANSFER WHILE FALSE DELETE	TWFD	95D2
TRANSFER WHILE FALSE UPDATE	TWFU	95DA
TRANSFER WHILE TRUE DELETE	TWTD	95D3
TRANSFER WHILE TRUE UPDATE	TWTU	95DB
TRANSFER WHILE LESS OR EQUAL UPDATE	TLEU	EB
TRANSFER WHILE LESS DELETE	TLSD	E0
TRANSFER WHILE LESS UPDATE	TLSU	E8
TRANSFER WHILE NOT EQUAL DELETE	TNED	E5
TRANSFER WHILE NOT EQUAL UPDATE	TNEU	ED
TRANSFER WORDS OVERWRITE DELETE	TWOD	D4
TRANSFER WORDS OVERWRITE UPDATE	TWOU	DC
TRANSFER WORDS DELETE	TWSD	D3

Table A-1. Operators, Alphabetical List (Cont)

Name	Mnemonic	Hexadecimal Code
TRANSFER WORDS UPDATE	TWSU	DB
TRANSLATE	TRNS	95D7
UNPACK ABSOLUTE DELETE	UABD	95D1
UNPACK ABSOLUTE UPDATE	UABU	95D9
UNPACK SIGNED DELETE	USND	95D0
UNPACKED SIGNED UPDATE	USNU	95D8
VALUE CALL	VALC	00 through 3F
WRITE TIME-OF-DAY	WTOD	9549

Table A-2. Operators, Numerical List

Hexadecimal Code	Name	Mnemonic
PRIMARY MODE		
00 thru 3F	VALUE CALL	VALC
40 thru 7F	NAME CALL	NAMC
80	ADD	ADD
81	SUBTRACT	SUBT
82	MULTIPLY	MULT
83	DIVIDE	DIVD
84	INTEGER DIVIDE	IDV
85	REMAINDER DIVIDE	RDIV
86	INTEGERIZE TRUNCATE	NTIA
87	INTEGERIZE ROUNDED	NTGR
88	LESS THAN	LESS
89	GREATER THAN OR EQUAL	GREQ
8A	GREATER THAN	GRTR
8B	LESS THAN OR EQUAL	LSEQ
8C	EQUAL	EQUL
8D	NOT EQUAL	NEQL
8E	CHANGE SIGN BIT	CHSN
8F	EXTENDED MULTIPLY	MULX
90	LOGICAL AND	LAND
91	LOGICAL OR	LOR
92	LOGICAL NEGATE	LNOT
93	LOGICAL EQUIVALENCE	LEQV
94	LOGICAL EQUAL	SAME
95	ESCAPE TO 16-BIT INSTRUCTIONS	VARI
96	BIT SET	BSET

Table A-2. Operators, Numerical List (Cont)

Hexadecimal Code	Name	Mnemonic
97	DYNAMIC BIT SET	DBST
98	FIELD TRANSFER	FLTR
99	DYNAMIC FIELD TRANSFER	DFTR
9A	FIELD ISOLATE	ISOL
9B	DYNAMIC FIELD ISOLATE	DISO
9C	FIELD INSERT	INSR
9D	DYNAMIC FIELD INSERT	DINS
9E	BIT RESET	BRST
9F	DYNAMIC BIT RESET	DBRS
A0	BRANCH FALSE	BRFL
A1	BRANCH TRUE	BRTR
A2	BRANCH UNCONDITIONAL	BRUN
A3	EXIT	EXIT
A5	INDEX AND LOAD NAME	NXLN
A6	INDEX	INDX
A7	RETURN	RETN
A8	DYNAMIC BRANCH FALSE	DBFL
A9	DYNAMIC BRANCH TRUE	DBTR
AA	DYNAMIC BRANCH UNCONDITIONAL	DBUN
AB	ENTER	ENTR
AC	EVALUATE DESCRIPTOR	EVAL
AD	INDEX AND LOAD VALUE	NXLV
AE	MARK STACK	MKST
AF	STUFF ENVIRONMENT	STFF
B0	LITERAL CALL ZERO	ZERO
B1	LITERAL CALL ONE	ONE
B2	LITERAL CALL 8-BITS	LT8
B3	LITERAL CALL 16-BITS	LT16
B4	PUSH DOWN STACK REGISTERS	PUSH
B5	DELETE TOP-OF-STACK	DLET
B6	EXCHANGE	EXCH
B7	DUPLICATE TOP-OF-STACK	DUPL
B8	STORE DELETE	STOD
B9	STORE NON-DELETE	STON
BA	OVERWRITE DELETE	OVRD
BB	OVERWRITE NON-DELETE	OVRN
BD	LOAD	LOAD
BE	LITERAL CALL 48-BITS	LT48
BF	MAKE PROGRAM CONTROL WORD	MPCW
C0	SCALE LEFT	SCLF
C1	DYNAMIC SCALE LEFT	DSLF
C2	SCALE RIGHT TRUNCATE	SCRT
C3	DYNAMIC SCALE RIGHT TRUNCATE	DSRT
C4	SCALE RIGHT SAVE	SCRS
C5	DYNAMIC SCALE RIGHT SAVE	DSRS
C6	SCALE RIGHT FINAL	SCRF
C7	DYNAMIC SCALE RIGHT FINAL	DSRF
C8	SCALE RIGHT ROUNDED	SCRR
C9	DYNAMIC SCALE RIGHT ROUNDED	DSRR
CA	INPUT CONVERT DELETE	ICVD

Table A-2. Operators, Numerical List (Cont)

Hexadecimal Code	Name	Mnemonic
CB	INPUT CONVERT UPDATE	ICVU
CC	SET TO SINGLE-PRECISION TRUNCATED	SNGT
CD	SET TO SINGLE-PRECISION ROUNDED	SNGL
CE	SET TO DOUBLE-PRECISION	XTND
CF	INSERT MARK STACK	IMKS
D0	TABLE ENTER EDIT DELETE	TEED
D1	PACK DESTRUCTIVE	PACD
D2	EXECUTE SINGLE MICRO DELETE	EXSD
D3	TRANSFER WORDS DESTRUCTIVE	TWSD
D4	TRANSFER WORDS OVERWRITE DELETE	TWOD
D5	STRING ISOLATE	SISO
D6	SET EXTERNAL SIGN	SXSN
D7	READ AND CLEAR OVERFLOW FLIP-FLOP	ROFF
D8	TABLE ENTER EDIT UPDATE	TEEU
D9	PACK UPDATE	PACU
DA	EXECUTE SINGLE MICRO UPDATE	EXSU
DB	TRANSFER WORDS UPDATE	TWSU
DC	TRANSFER WORDS OVERWRITE UPDATE	TWOU
DD	EXECUTE SINGLE MICRO SINGLE POINTER UPDATE	EXPU
DE	READ TRUE/FALSE FLIP-FLOP	TRFF
DF	CONDITIONAL HALT	HALT
E0	TRANSFER WHILE LESS DELETE	TLSD
E1	TRANSFER WHILE GREATER OR EQUAL DELETE	TGED
E2	TRANSFER WHILE GREATER DELETE	TGTD
E3	TRANSFER WHILE LESS OR EQUAL DELETE	TLED
E4	TRANSFER WHILE EQUAL DELETE	TEQD
E5	TRANSFER WHILE NOT EQUAL DELETE	TNED
E6	TRANSFER UNCONDITIONAL DELETE	TUND
E8	TRANSFER WHILE LESS UPDATE	TLSU
E9	TRANSFER WHILE GREATER OR EQUAL UPDATE	TGEU
EA	TRANSFER WHILE GREATER UPDATE	TGTU
EB	TRANSFER WHILE LESS OR EQUAL UPDATE	TLEU
EC	TRANSFER WHILE EQUAL UPDATE	TEQU
ED	TRANSFER WHILE NOT EQUAL UPDATE	TNEU
EE	TRANSFER UNCONDITIONAL UPDATE	TUNU
F0	COMPARE CHARACTERS LESS DELETE	CLSD
F1	COMPARE CHARACTERS GREATER OR EQUAL DELETE	CGED
F2	COMPARE CHARACTERS GREATER DELETE	CGTD
F3	COMPARE CHARACTERS LESS OR EQUAL DELETE	CLED
F4	COMPARE CHARACTERS EQUAL DELETE	CEQD
F5	COMPARE CHARACTERS NOT EQUAL DELETE	CNED
F8	COMPARE CHARACTERS LESS UPDATE	CLSU
F9	COMPARE CHARACTERS GREATER OR EQUAL UPDATE	CGEU
FA	COMPARE CHARACTERS GREATER UPDATE	CGTU
FB	COMPARE CHARACTERS LESS OR EQUAL UPDATE	CLEU
FC	COMPARE CHARACTERS EQUAL UPDATE	CEQU
FD	COMPARE CHARACTERS NOT EQUAL UPDATE	CNEU
FE	NO OPERATION	NOOP
FF	INVALID OPERATOR	NVLD

Table A-2. Operators, Numerical List (Cont)

Hexadecimal Code	Name	Mnemonic
VARIANT MODE		
9541	RUNNING INDICATOR	RUNI
9542	SET TWO SINGLES TO DOUBLE	JOIN
9543	SET DOUBLE TO TWO SINGLES	SPLT
9545	SET INTERVAL TIMER	SINT
9546	ENABLE EXTERNAL INTERRUPTS	EEXI
9547	DISABLE EXTERNAL INTERRUPTS	DEXI
9549	WRITE TIME-OF-DAY	WTOD
954A	SCAN-IN	SCNI
954B	SCAN-OUT	SCNO
954C	COMMUNICATE WITH UNIVERSAL I/O	CUIO
954E	READ PROCESSOR IDENTIFICATION	WHOI
9585	OCCURS INDEX	OCRX
9587	INTEGERIZE, ROUNDED, DOUBLE-PRECISION	NTGD
958B	LEADING ONE TEST	LOG2
958E	NORMALIZE	NORM
95A7	READ TIME-OF-DAY	RTOD
95AE	MOVE TO STACK	MVST
95B4	SET TAG FIELD	STAG
95B5	READ TAG FIELD	RTAG
95B6	ROTATE STACK UP	RSUP
95B7	ROTATE STACK DOWN	RSDN
95B8	READ PROCESSOR REGISTER	RPRR
95B9	SET PROCESSOR REGISTER	SPRR
95BA	READ WITH LOCK	RDLK
95BB	COUNT BINARY ONES	CBON
95BC	LOAD TRANSPARENT	LODT
95BD	LINKED LIST LOOKUP	LLLU
95BE	MASKED SEARCH FOR EQUAL	SRCH
95D0	UNPACK SIGNED DELETE	USND
95D1	UNPACK ABSOLUTE DELETE	UABD
95D2	TRANSFER WHILE FALSE DELETE	TWFD
95D3	TRANSFER WHILE TRUE DELETE	TWTD
95D4	SCAN WHILE FALSE DELETE	SWFD
95D5	SCAN WHILE TRUE DELETE	SWTD
95D7	TRANSLATE	TRNS
95D8	UNPACK SIGNED UPDATE	USNU
95D9	UNPACK ABSOLUTE UPDATE	UABU
95DA	TRANSFER WHILE FALSE UPDATE	TWFU
95DB	TRANSFER WHILE TRUE UPDATE	TWTU
95DC	SCAN WHILE FALSE UPDATE	SWFU
95DD	SCAN WHILE TRUE UPDATE	SWTU
95DF	CONDITIONAL HALT	HALT
95F0	SCAN WHILE LESS DELETE	SLSD
95F1	SCAN WHILE GREATER OR EQUAL DELETE	SGED
95F2	SCAN WHILE GREATER DELETE	SGTD
95F3	SCAN WHILE LESS OR EQUAL DELETE	SLED
95F4	SCAN WHILE EQUAL DELETE	SEQD
95F5	SCAN WHILE NOT EQUAL DELETE	SNED
95F8	SCAN WHILE LESS UPDATE	SLSU
95F9	SCAN WHILE GREATER OR EQUAL UPDATE	SGEU

Table A-2. Operators, Numerical List (Cont)

Hexadecimal Code	Name	Mnemonic
95FA	SCAN WHILE GREATER UPDATE	SGTU
95FB	SCAN WHILE LESS OR EQUAL UPDATE	SLEU
95FC	SCAN WHILE EQUAL UPDATE	SEQU
95FD	SCAN WHILE NOT EQUAL UPDATE	SNEU
95FE	NO OPERATION	NOOP
95FF	INVALID OPERATOR	NVLD
EDIT MODE		
D0	MOVE WITH INSERT	MINS
D1	MOVE WITH FLOAT	MFLT
D2	SKIP FORWARD SOURCE CHARACTERS	SFSC
D3	SKIP REVERSE SOURCE CHARACTERS	SRSC
D4	RESET FLOAT	RSTF
D5	END FLOAT	ENDF
D6	MOVE NUMERIC UNCONDITIONAL	MVNU
D7	MOVE CHARACTERS	MCHR
D8	INSERT OVERPUNCH	INOP
D9	INSERT DISPLAY SIGN	INSG
DA	SKIP FORWARD DESTINATION CHARACTERS	SFDC
DB	SKIP REVERSE DESTINATION CHARACTERS	SRDC
DC	INSERT UNCONDITIONAL	INSU
DD	INSERT CONDITIONAL	INSC
DE	END EDIT	ENDE
DF	CONDITIONAL HALT	HALT
FE	NO OPERATION	NOOP
FF	INVALID OPERATOR	NVLD

APPENDIX B
DATA REPRESENTATION

Table B-1. Data Representation

EBCDIC Graphic	Decimal Value	EBCDIC Internal	Hex. Graphic	EBCDIC Card Code	Octal
BLANK	64	0100 0000	40	No Punches	60
[	74	0100 1010	4A	12 8 2	33
.	75	0100 1011	4B	12 8 3	32
<	76	0100 1100	4C	12 8 4	36
(	77	0100 1101	4D	12 8 5	35
+	78	0100 1110	4E	12 8 6	
|	79	0100 1111	4F	12 8 7	37
&	80	0101 0000	50	12	34
]	90	0101 1010	5A	11 8 2	76
$	91	0101 1011	5B	11 8 3	52
*	92	0101 1100	5C	11 8 4	53
)	93	0101 1101	5D	11 8 5	55
;	94	0101 1110	5E	11 8 6	56
	95	0101 1111	5F	11 8 7	57
-	96	0110 0000	60	11	54
/	97	0110 0001	61	0 1	61
,	107	0110 1011	6B	0 8 3	72
%	108	0110 1100	6C	0 8 4	73
_	109	0110 1101	6D	0 8 5	74
>	110	0110 1110	6E	0 8 6	16
?	111	0110 1111	6F	0 8 7	14
:	122	0111 1010	7A	8 2	15
#	123	0111 1011	7B	8 3	12
@	124	0111 1100	7C	8 4	13
'	125	0111 1101	7D	8 5	17
=	126	0111 1110	7E	8 6	75
"	127	0111 1111	7F	8 7	77
(+)PZ	192	1100 0000	C0	12 0	20
A	193	1100 0001	C1	12 1	21
B	194	1100 0010	C2	12 2	22
C	195	1100 0011	C3	12 3	23
D	196	1100 0100	C4	12 4	24
E	197	1100 0101	C5	12 5	25
F	198	1100 0110	C6	12 6	26
G	199	1100 0111	C7	12 7	27
H	200	1100 1000	C8	12 8	30
I	201	1100 1001	C9	12 9	31
(!)MZ	208	1101 0000	D0	11 0	40
J	209	1101 0001	D1	11 1	41
K	210	1101 0010	D2	11 2	42
L	211	1101 0011	D3	11 3	43
M	212	1101 0100	D4	11 4	44

Table B-1. Data Representation (Cont)

EBCDIC Graphic	Decimal Value	EBCDIC Internal	Hex. Graphic	EBCDIC Card Code	Octal
N	213	1101 0101	D5	11 5	45
O	214	1101 0110	D6	11 6	46
P	215	1101 0111	D7	11 7	47
Q	216	1101 1000	D8	11 8	50
R	217	1101 1001	D9	11 9	51
_	224	1110 0000	E0	0 8 2	
S	226	1110 0010	E2	0 2	62
T	227	1110 0011	E3	0 3	63
U	228	1110 0100	E4	0 4	64
V	229	1110 0101	E5	0 5	65
W	230	1110 0110	E6	0 6	66
X	231	1110 0111	E7	0 7	67
Y	232	1110 1000	E8	0 8	70
Z	233	1110 1001	E9	0 9	71
0	240	1111 0000	F0	0	00
1	241	1111 0001	F1	1	01
2	242	1111 0010	F2	2	02
3	243	1111 0011	F3	3	03
4	244	1111 0100	F4	4	04
5	245	1111 0101	F5	5	05
6	246	1111 0110	F6	6	06
7	247	1111 0111	F7	7	07
8	248	1111 1000	F8	8	10
9	249	1111 1001	F9	9	11

Documentation Evaluation Form

Title: B 5900 System Reference Manual

Form No: 5011034

Date: September 1981

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

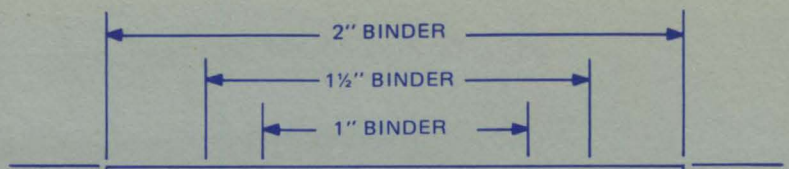
Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
Documentation Dept., TIO - West
1300 John Reed Court
City of Industry, CA 91745
U.S.A.



B 5900 System

REFERENCE MANUAL

5011034

Printed in U.S.A.